(12) UK Patent Application (19) GB (11) 2 337 140 (13) A

(43) Date of A Publication 10.11.1999

(21) Application No 9819966.4

(22) Date of Filing 15.09.1998

(30) Priority Data
(31) 2236640 (32) 04.05.1998 (33) CA

(71) Applicant(s)
Mitel Corporation
(Incorporated in Canada - Ontario)
P O Box 13089, Kanata, Ontario K2K 1X3, Canada

(72) Inventor(s)
Dino Canton
Claudio Gambetti

(74) Agent and/or Address for Service
Cruikshank & Fairweather
19 Royal Exchange Square, GLASGOW, G1 3AE,
United Kingdom

(51) INT CL[6]
G06F 9/455

(52) UK CL (Edition Q)
G4A AFP
U1S S2214

(56) Documents Cited
GB 2203572 A **GB 1593780 A** **GB 1497601 A**
GB 1485874 A **US 4691278 A**

(58) Field of Search
UK CL (Edition Q) **G4A AFP**
INT CL[6] **G06F**

(54) Abstract Title
Emulation in microprocessors

(57) An emulation system to allow code written for a specific computing environment to be run on a different microprocessor platform comprises:

an emulation routine generator including a table storing code segments corresponding to instructions of said specific computing environment, said emulation routine generator creating said emulation routines written in code for said different microprocessor platform from selected code segments in said table, said emulation routines corresponding to instructions of said specific computing environment;

an emulator kernel to read instructions of said specific computing environment and access and execute emulation routines thereby to emulate and carry out said read instructions; and

an interface resembling said specific computing environment.

The generator also puts linking addresses to the emulation routines into a jump table.

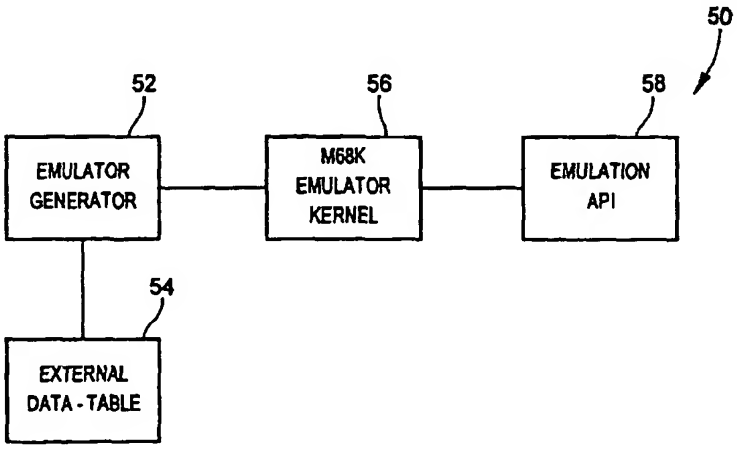


FIG. 1

GB 2 337 140 A

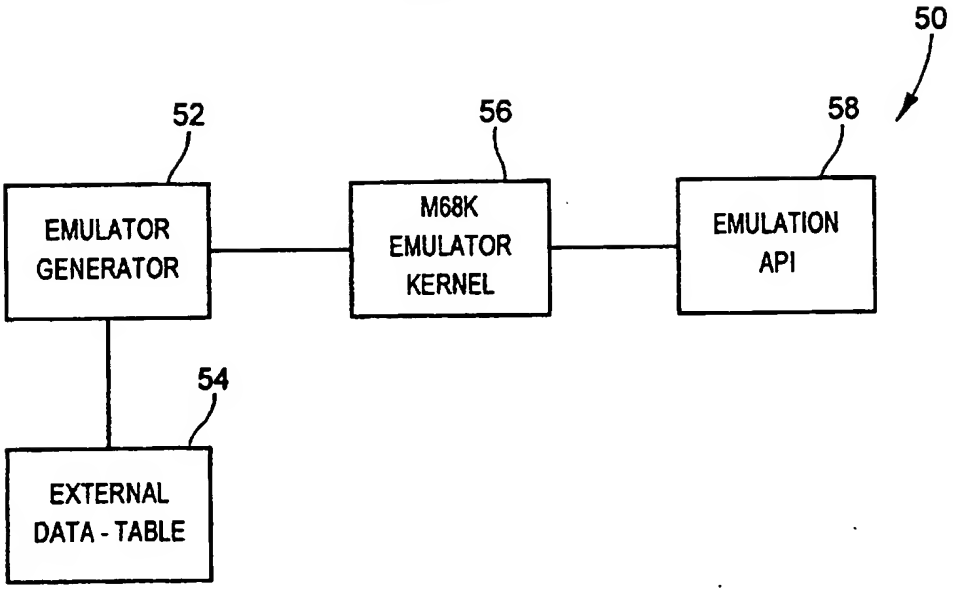

50
52
56
58
EMULATOR GENERATOR
M68K EMULATOR KERNEL
EMULATION API
54
EXTERNAL DATA - TABLE

FIG. 1

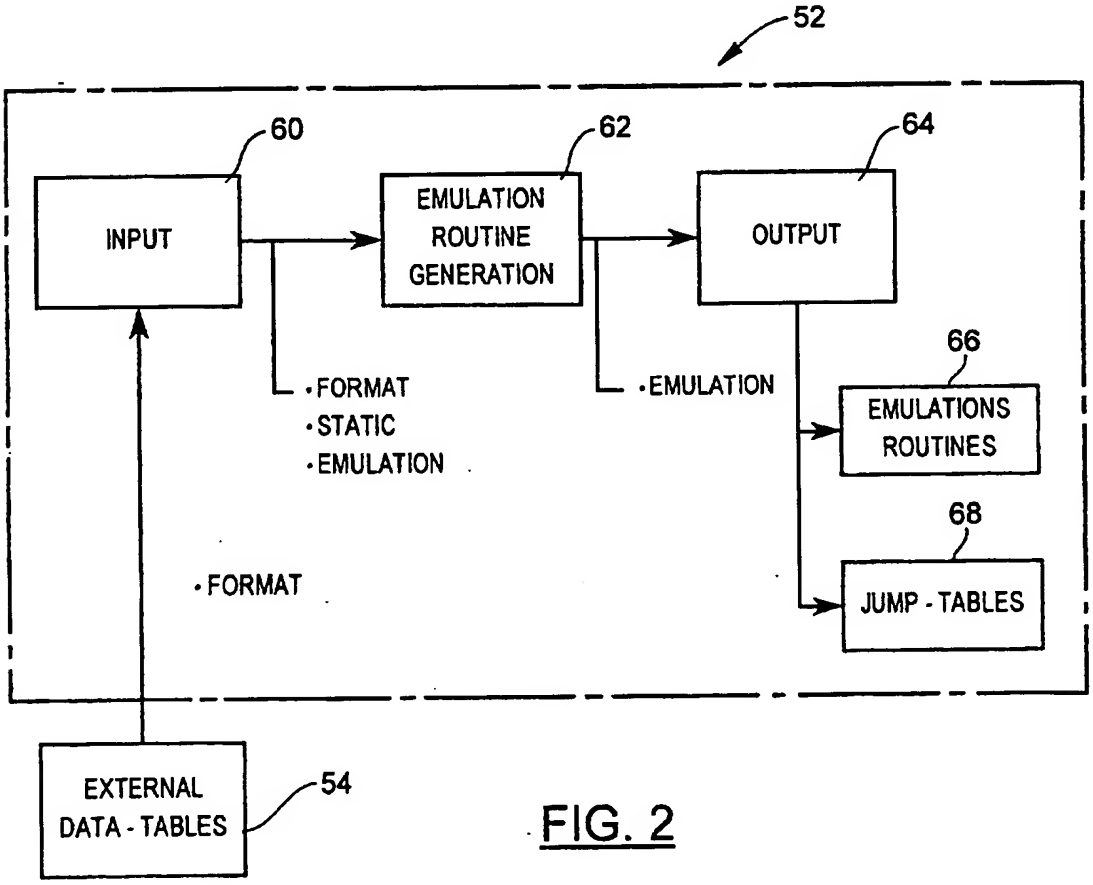

52
60
62
64
INPUT
EMULATION ROUTINE GENERATION
OUTPUT
•FORMAT
•STATIC
•EMULATION
•EMULATION
66
EMULATIONS ROUTINES
68
JUMP - TABLES
•FORMAT
EXTERNAL DATA - TABLES
54

FIG. 2

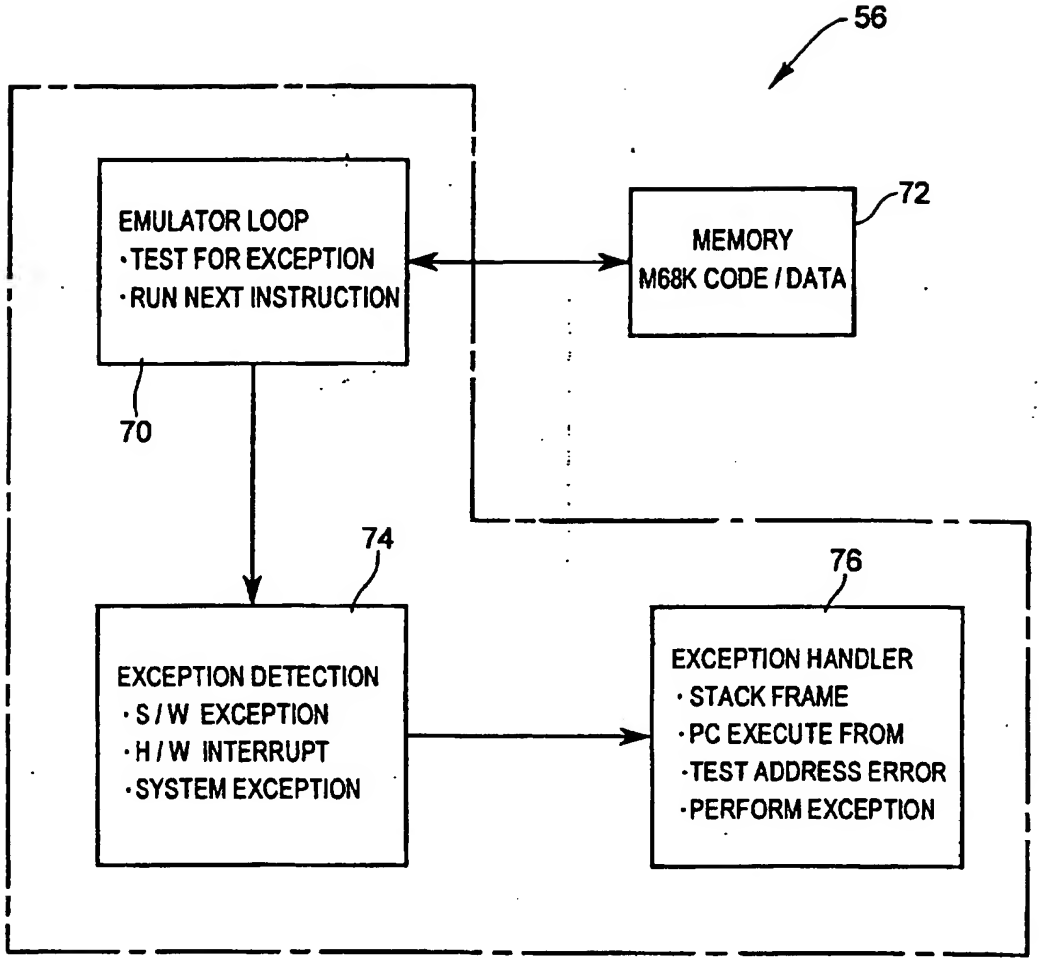

56
EMULATOR LOOP
·TEST FOR EXCEPTION
·RUN NEXT INSTRUCTION
70
MEMORY
M68K CODE / DATA
72
74
EXCEPTION DETECTION
· S / W EXCEPTION
· H / W INTERRUPT
· SYSTEM EXCEPTION
76
EXCEPTION HANDLER
· STACK FRAME
· PC EXECUTE FROM
· TEST ADDRESS ERROR
· PERFORM EXCEPTION

FIG. 3

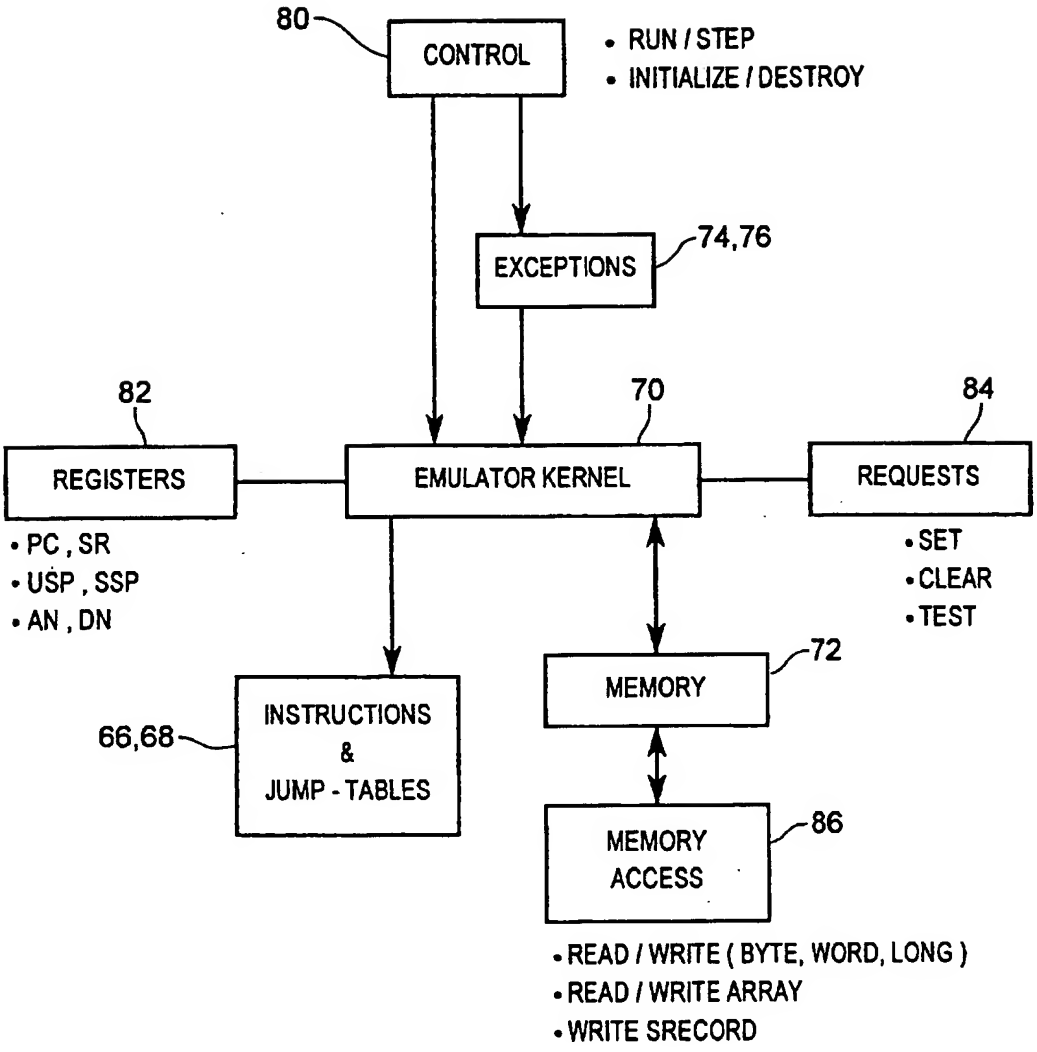

80
CONTROL
• RUN / STEP
• INITIALIZE / DESTROY
EXCEPTIONS
74,76
82
70
84
REGISTERS
EMULATOR KERNEL
REQUESTS
• PC , SR
• USP , SSP
• AN , DN
• SET
• CLEAR
• TEST
72
MEMORY
66,68
INSTRUCTIONS
&
JUMP - TABLES
MEMORY
ACCESS
86
• READ / WRITE (BYTE, WORD, LONG)
• READ / WRITE ARRAY
• WRITE SRECORD

FIG. 4

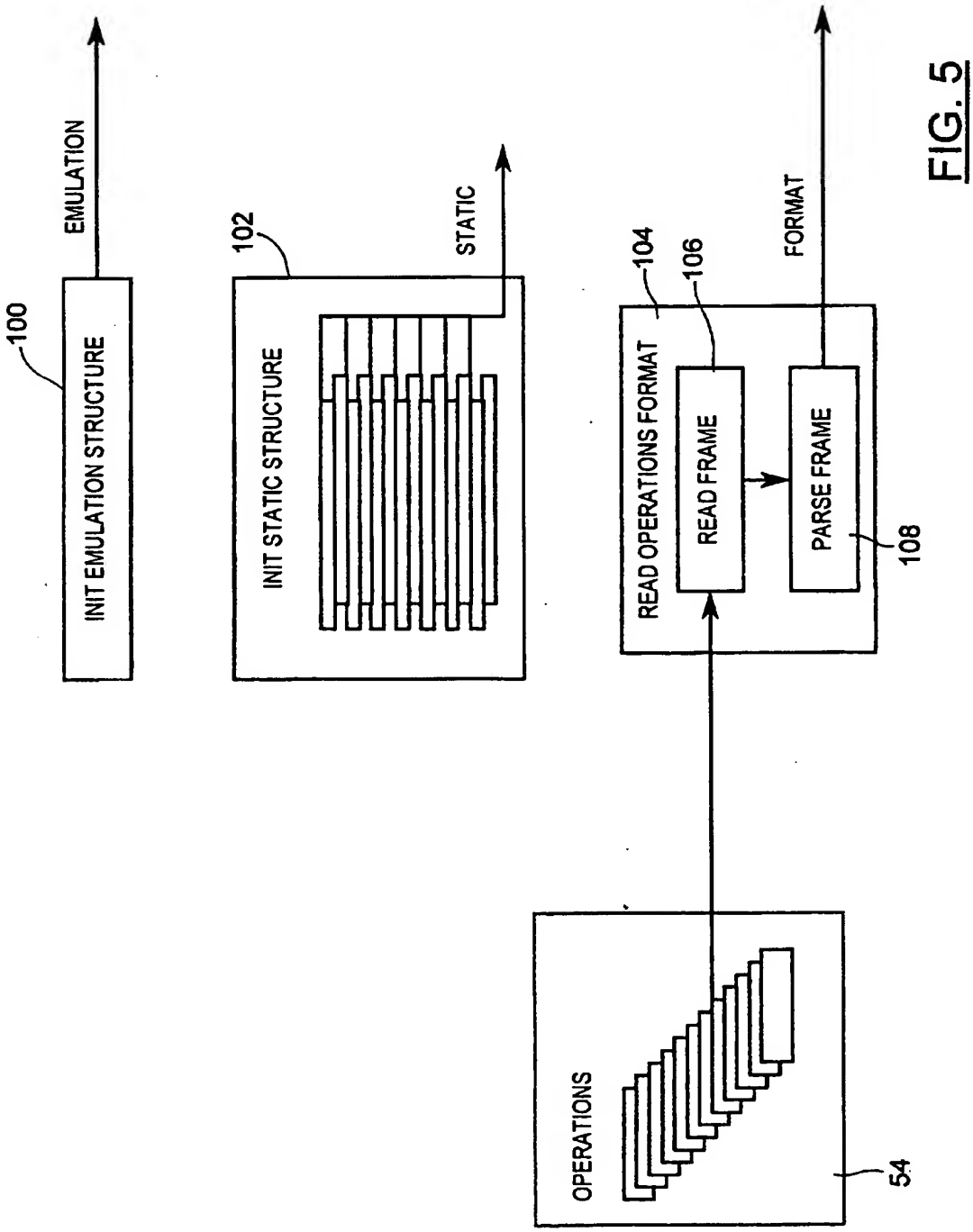

EMULATION
100
INIT EMULATION STRUCTURE
102
STATIC
INIT STATIC STRUCTURE
104
106
FORMAT
READ OPERATIONS FORMAT
READ FRAME
PARSE FRAME
108
OPERATIONS
54

FIG. 5

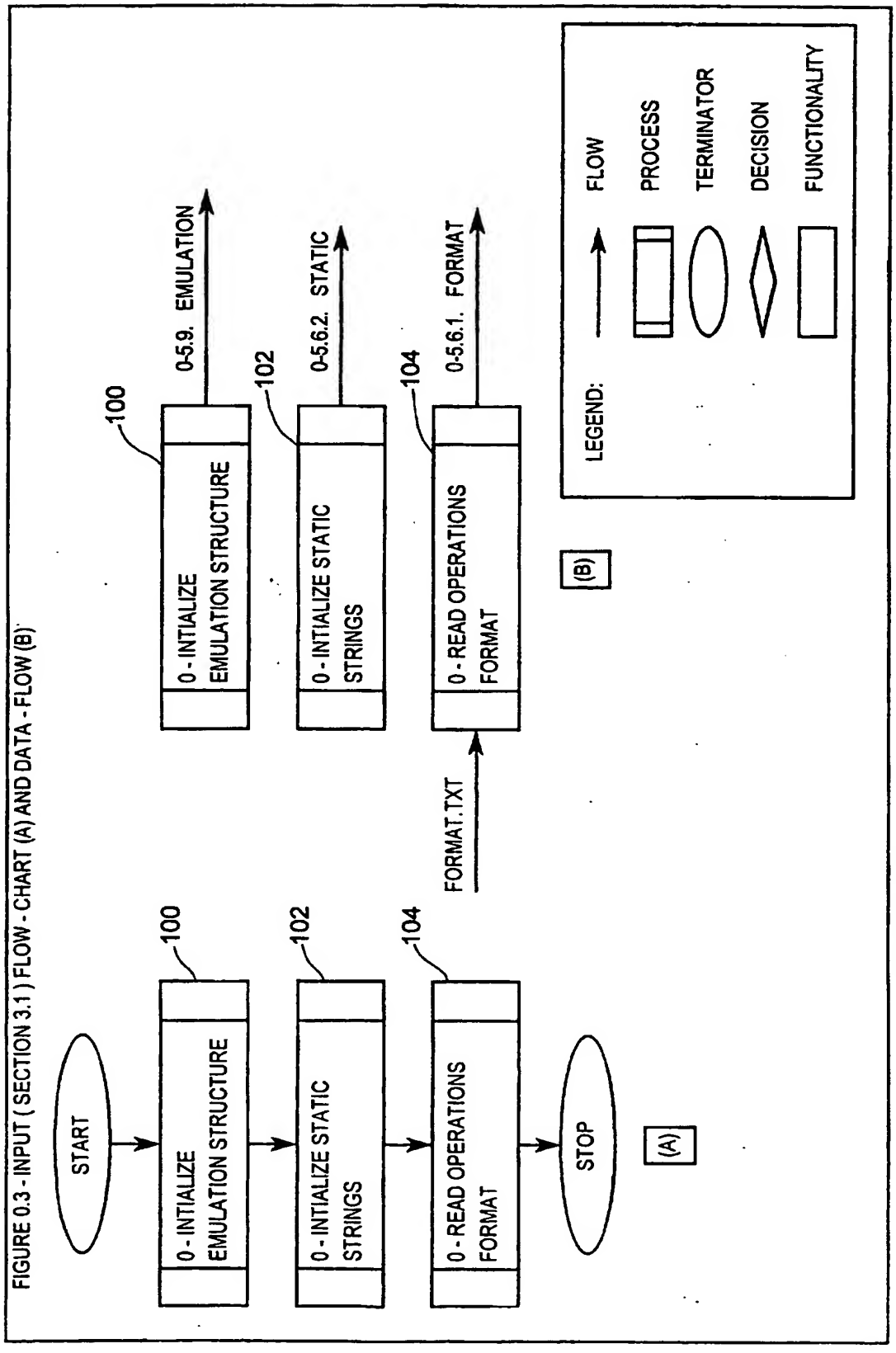


FIGURE 0.3 - INPUT (SECTION 3.1) FLOW - CHART (A) AND DATA - FLOW (B)
START
0 - INTIALIZE EMULATION STRUCTURE
0 - INTIALIZE STATIC STRINGS
0 - READ OPERATIONS FORMAT
STOP
(A)
100
102
104
FORMAT.TXT
0-5.9. EMULATION
0-5.6.2. STATIC
0-5.6.1. FORMAT
(B)
LEGEND:
FLOW
PROCESS
TERMINATOR
DECISION
FUNCTIONALITY

FIG. 6

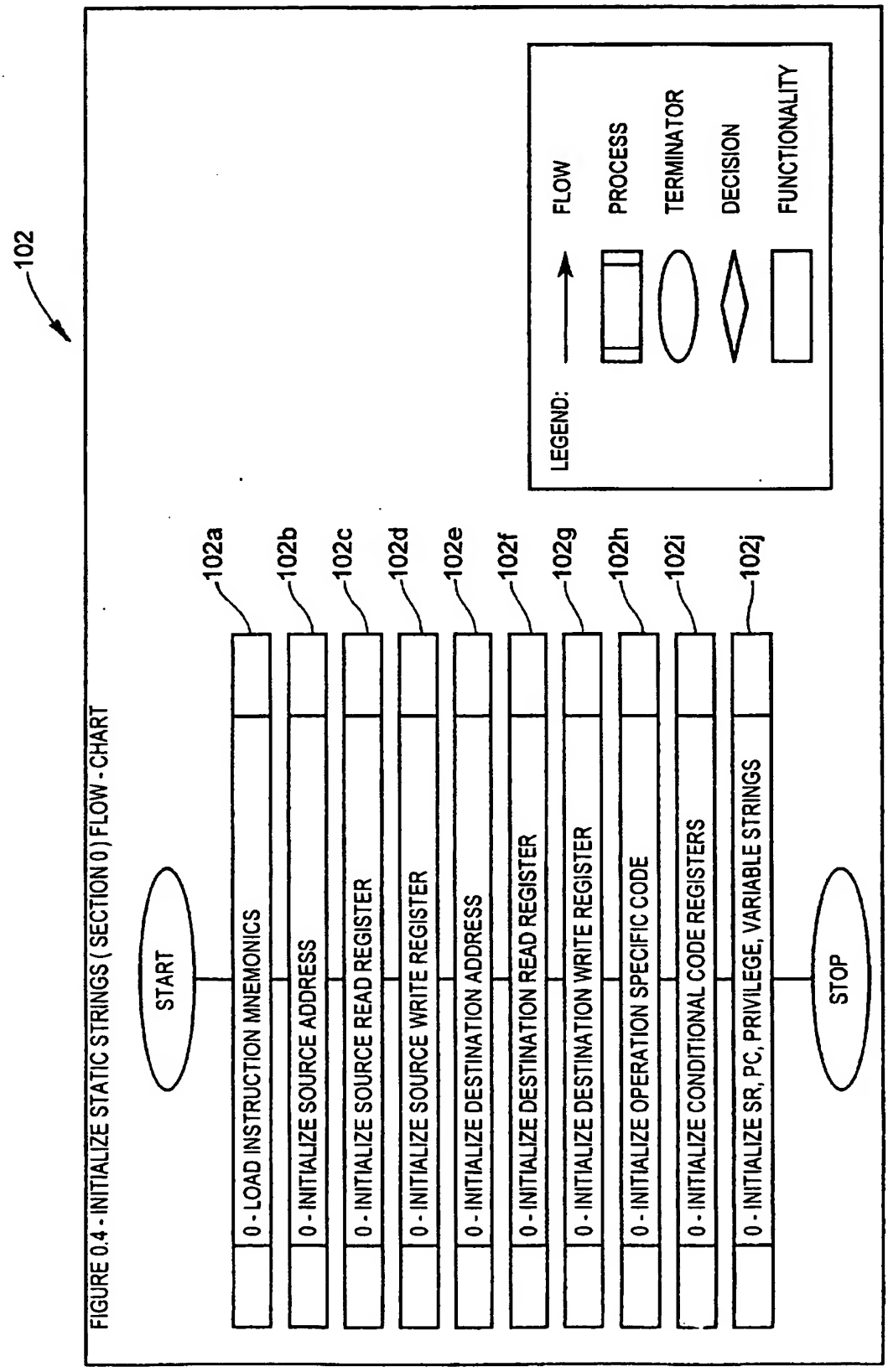

102
FIGURE 0.4 - INITIALIZE STATIC STRINGS (SECTION 0) FLOW - CHART
START
0 - LOAD INSTRUCTION MNEMONICS
0 - INITIALIZE SOURCE ADDRESS
0 - INITIALIZE SOURCE READ REGISTER
0 - INITIALIZE SOURCE WRITE REGISTER
0 - INITIALIZE DESTINATION ADDRESS
0 - INITIALIZE DESTINATION READ REGISTER
0 - INITIALIZE DESTINATION WRITE REGISTER
0 - INITIALIZE OPERATION SPECIFIC CODE
0 - INITIALIZE CONDITIONAL CODE REGISTERS
0 - INITIALIZE SR, PC, PRIVILEGE, VARIABLE STRINGS
STOP
102a
102b
102c
102d
102e
102f
102g
102h
102i
102j
LEGEND:
FLOW
PROCESS
TERMINATOR
DECISION
FUNCTIONALITY

FIG. 7

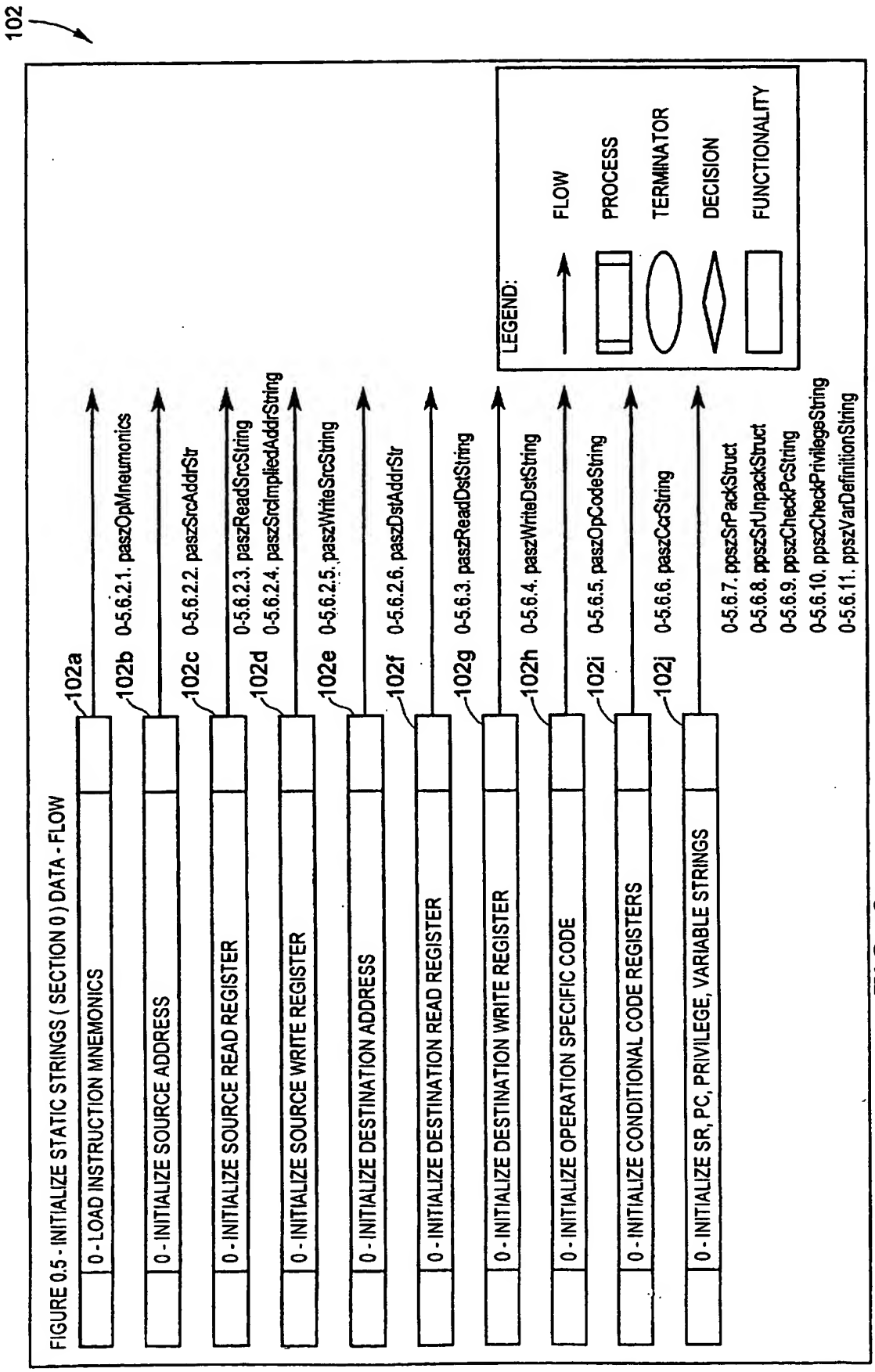

102
FIGURE 0.5 - INITIALIZE STATIC STRINGS (SECTION 0) DATA - FLOW
0 - LOAD INSTRUCTION MNEMONICS
102a
0 - INITIALIZE SOURCE ADDRESS
102b
0-5.6.2.1. paszOpMneumonics
0 - INITIALIZE SOURCE READ REGISTER
102c
0-5.6.2.2. paszSrcAddrStr
0 - INITIALIZE SOURCE WRITE REGISTER
102d
0-5.6.2.3. paszReadSrcString
0-5.6.2.4. paszSrcImpliedAddrString
0 - INITIALIZE DESTINATION ADDRESS
102e
0-5.6.2.5. paszWriteSrcString
0 - INITIALIZE DESTINATION READ REGISTER
102f
0-5.6.2.6. paszDstAddrStr
0 - INITIALIZE DESTINATION WRITE REGISTER
102g
0-5.6.3. paszReadDstString
0 - INITIALIZE OPERATION SPECIFIC CODE
102h
0-5.6.4. paszWriteDstString
0 - INITIALIZE CONDITIONAL CODE REGISTERS
102i
0-5.6.5. paszOpCodeString
0 - INITIALIZE SR, PC, PRIVILEGE, VARIABLE STRINGS
102j
0-5.6.6. paszCcString
0-5.6.7. ppszSrPackStruct
0-5.6.8. ppszSrUnpackStruct
0-5.6.9. ppszCheckPcString
0-5.6.10. ppszCheckPrivilegeString
0-5.6.11. ppszVarDefinitionString
LEGEND:
FLOW
PROCESS
TERMINATOR
DECISION
FUNCTIONALITY

FIG. 8

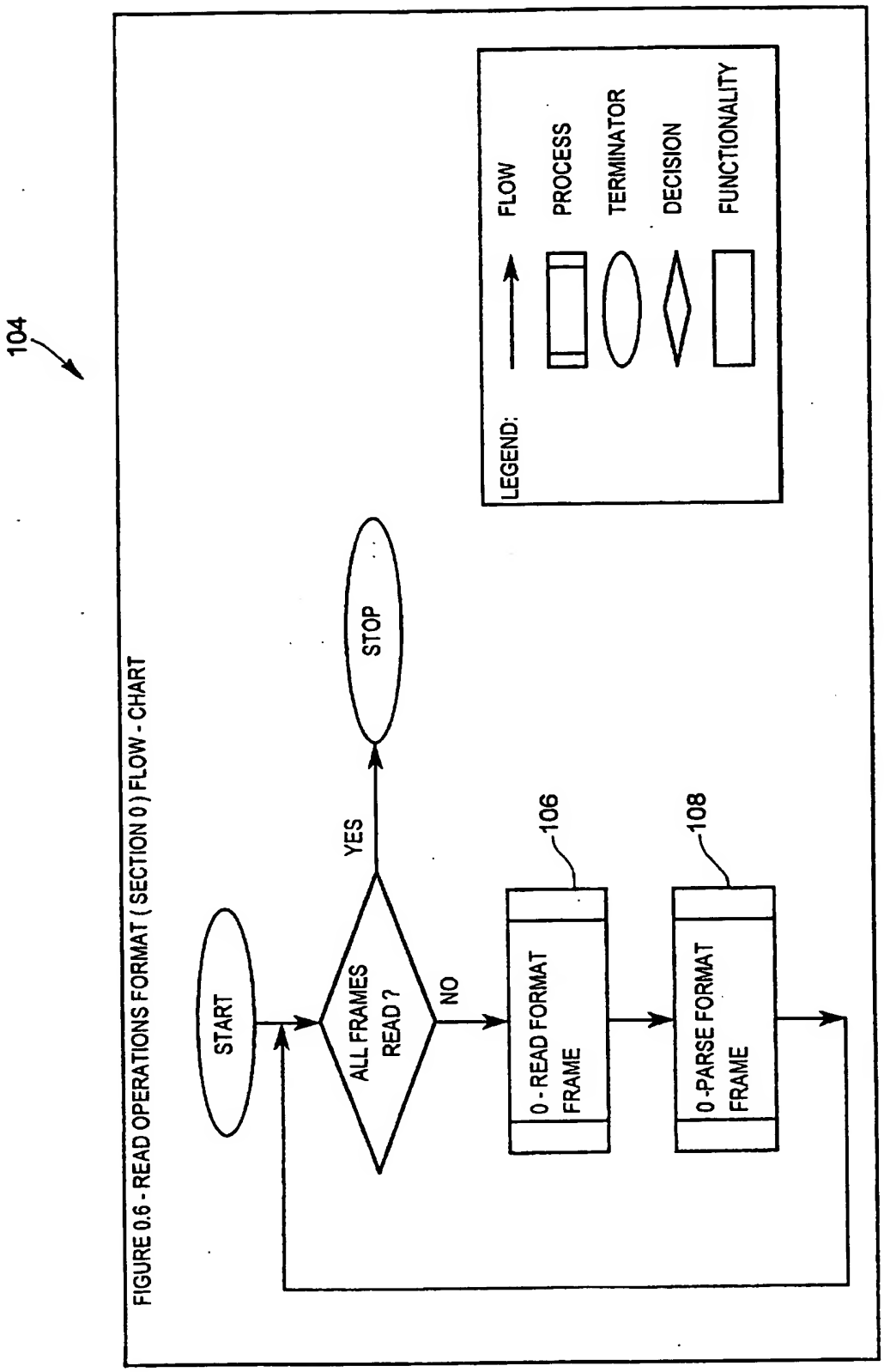

104
FIGURE 0.6 - READ OPERATIONS FORMAT (SECTION 0) FLOW - CHART
START
ALL FRAMES READ ?
YES
NO
STOP
0 - READ FORMAT FRAME
106
0 -PARSE FORMAT FRAME
108
LEGEND:
FLOW
PROCESS
TERMINATOR
DECISION
FUNCTIONALITY

FIG. 9

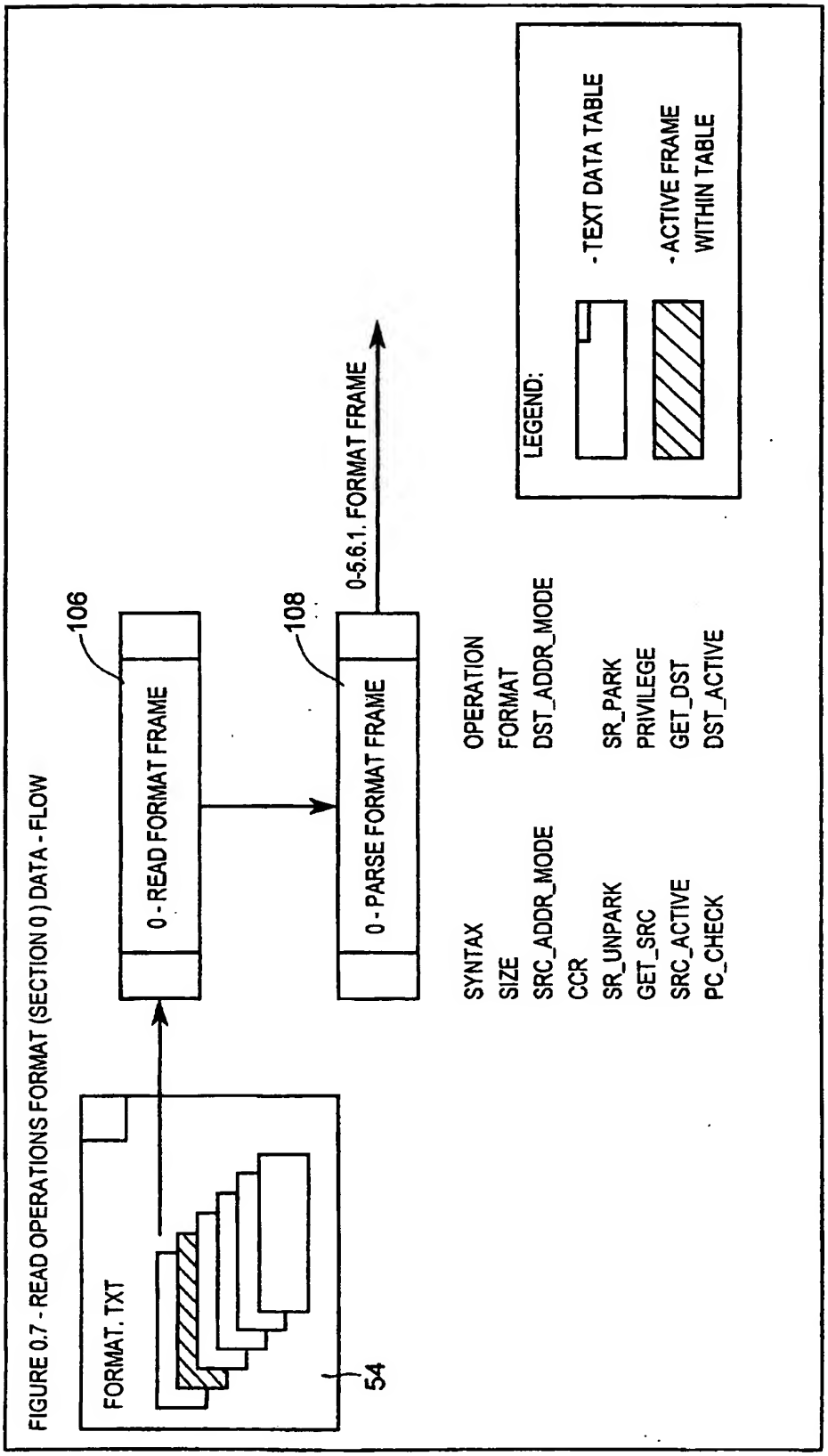


FIGURE 0.7 - READ OPERATIONS FORMAT (SECTION 0) DATA - FLOW
FORMAT. TXT
54
0 - READ FORMAT FRAME
106
0 - PARSE FORMAT FRAME
108
0-5.6.1. FORMAT FRAME
SYNTAX
SIZE
SRC_ADDR_MODE
CCR
SR_UNPARK
GET_SRC
SRC_ACTIVE
PC_CHECK
OPERATION
FORMAT
DST_ADDR_MODE
SR_PARK
PRIVILEGE
GET_DST
DST_ACTIVE
LEGEND:
- TEXT DATA TABLE
- ACTIVE FRAME WITHIN TABLE

FIG. 10

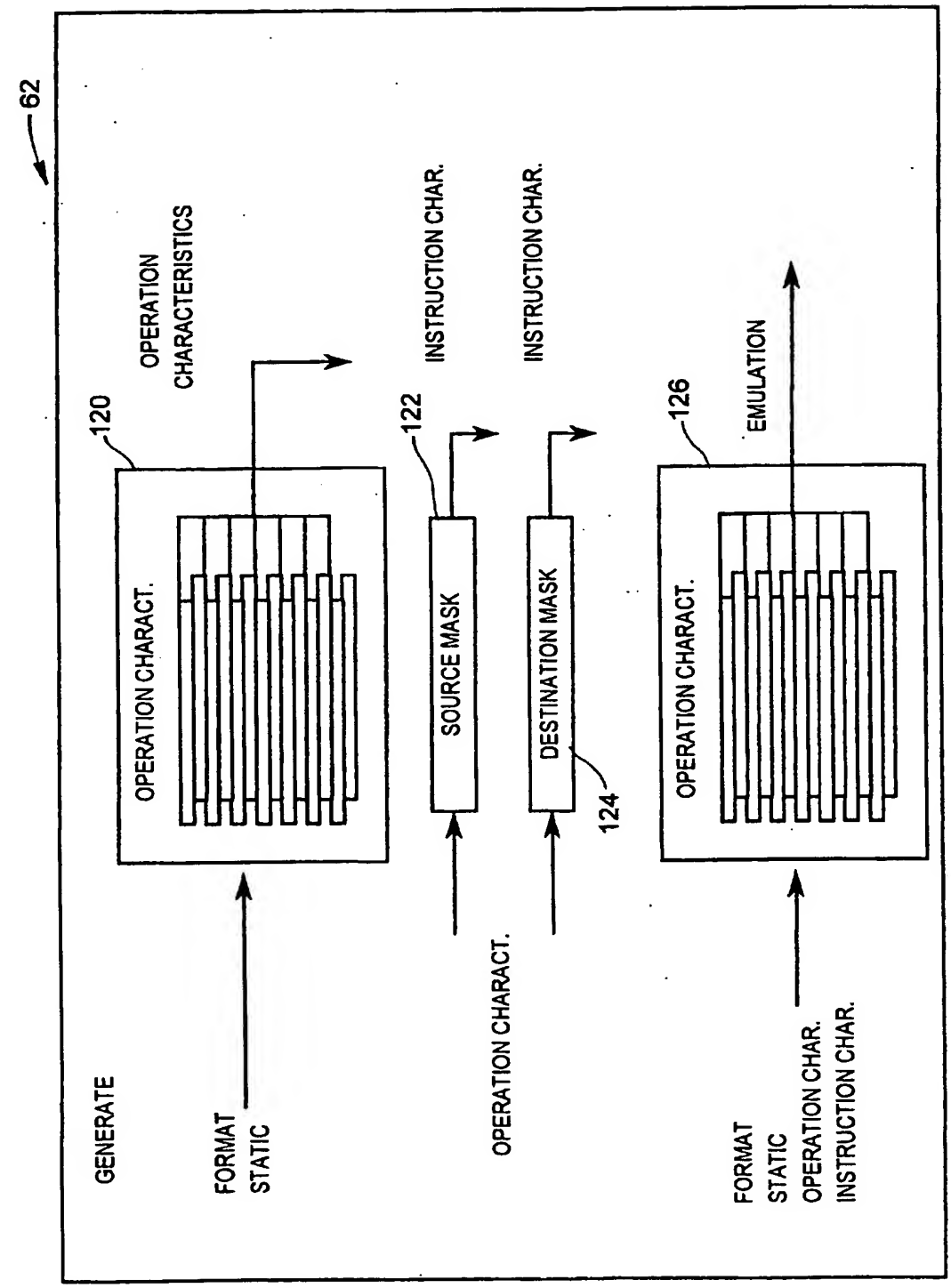

62
GENERATE
FORMAT
STATIC
OPERATION CHARACTERISTICS
120
OPERATION CHARACT.
OPERATION CHARACT.
122
SOURCE MASK
INSTRUCTION CHAR.
124
DESTINATION MASK
INSTRUCTION CHAR.
126
OPERATION CHARACT.
EMULATION
FORMAT
STATIC
OPERATION CHAR.
INSTRUCTION CHAR.

FIG. 11

FIGURE 0.9 - 5.4. GENERATION (SECTION 0) FLOW - CHART

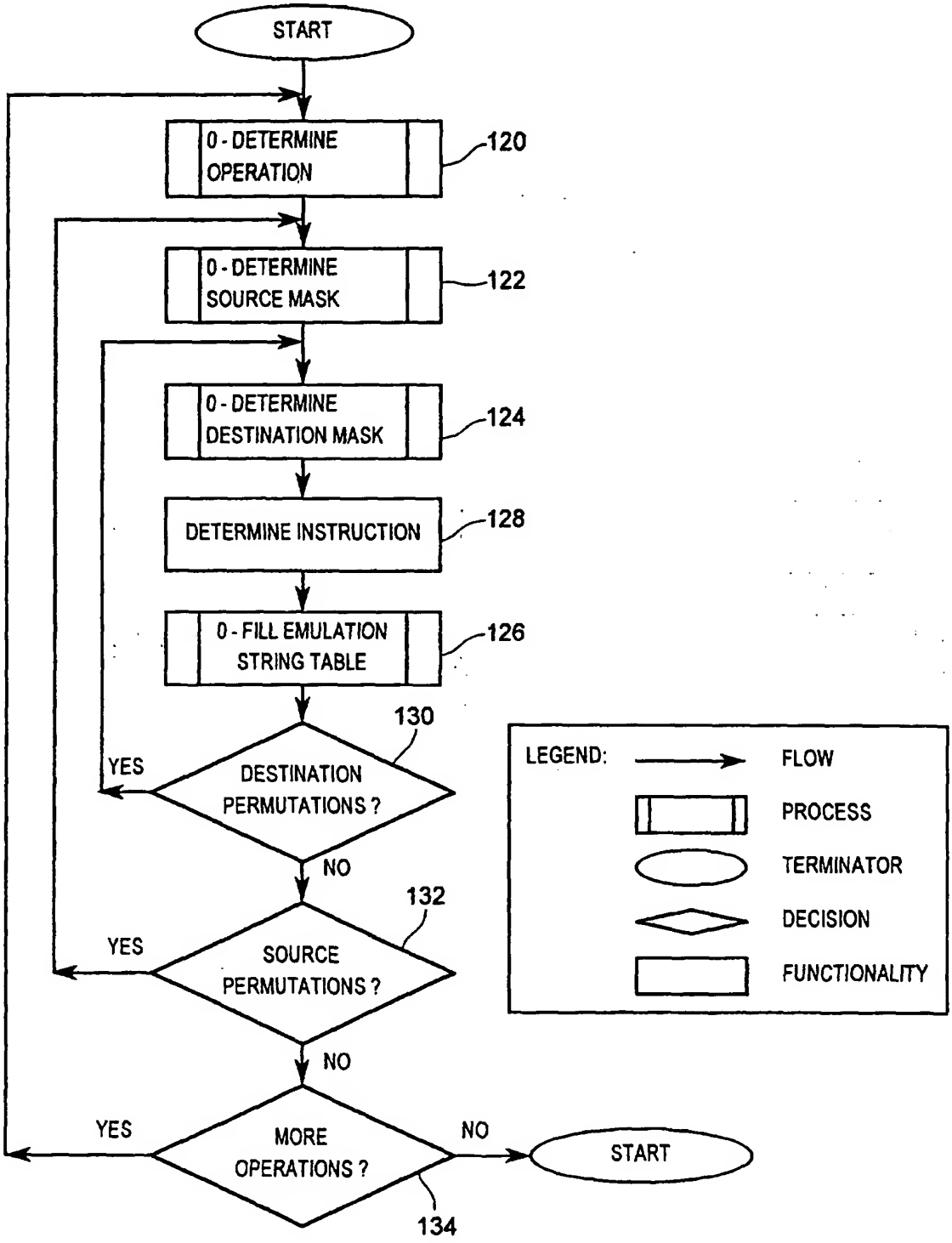


FIG. 12

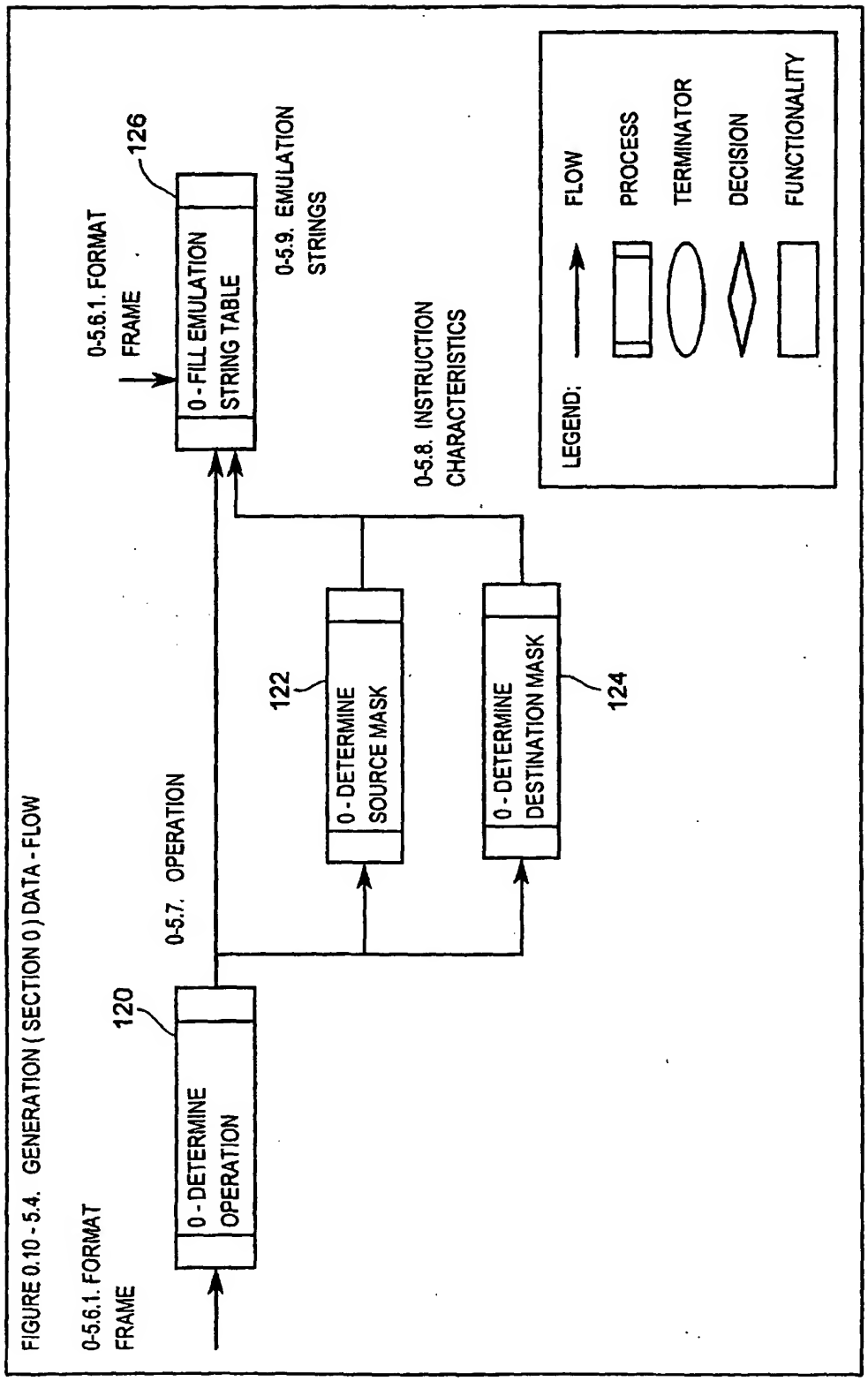


FIGURE 0.10 - 5.4. GENERATION (SECTION 0) DATA - FLOW
0-5.6.1. FORMAT FRAME
0 - DETERMINE OPERATION
120
0-5.7. OPERATION
0 - DETERMINE SOURCE MASK
122
0 - DETERMINE DESTINATION MASK
124
0-5.8. INSTRUCTION CHARACTERISTICS
0-5.6.1. FORMAT FRAME
0 - FILL EMULATION STRING TABLE
126
0-5.9. EMULATION STRINGS
LEGEND:
FLOW
PROCESS
TERMINATOR
DECISION
FUNCTIONALITY

FIG. 13

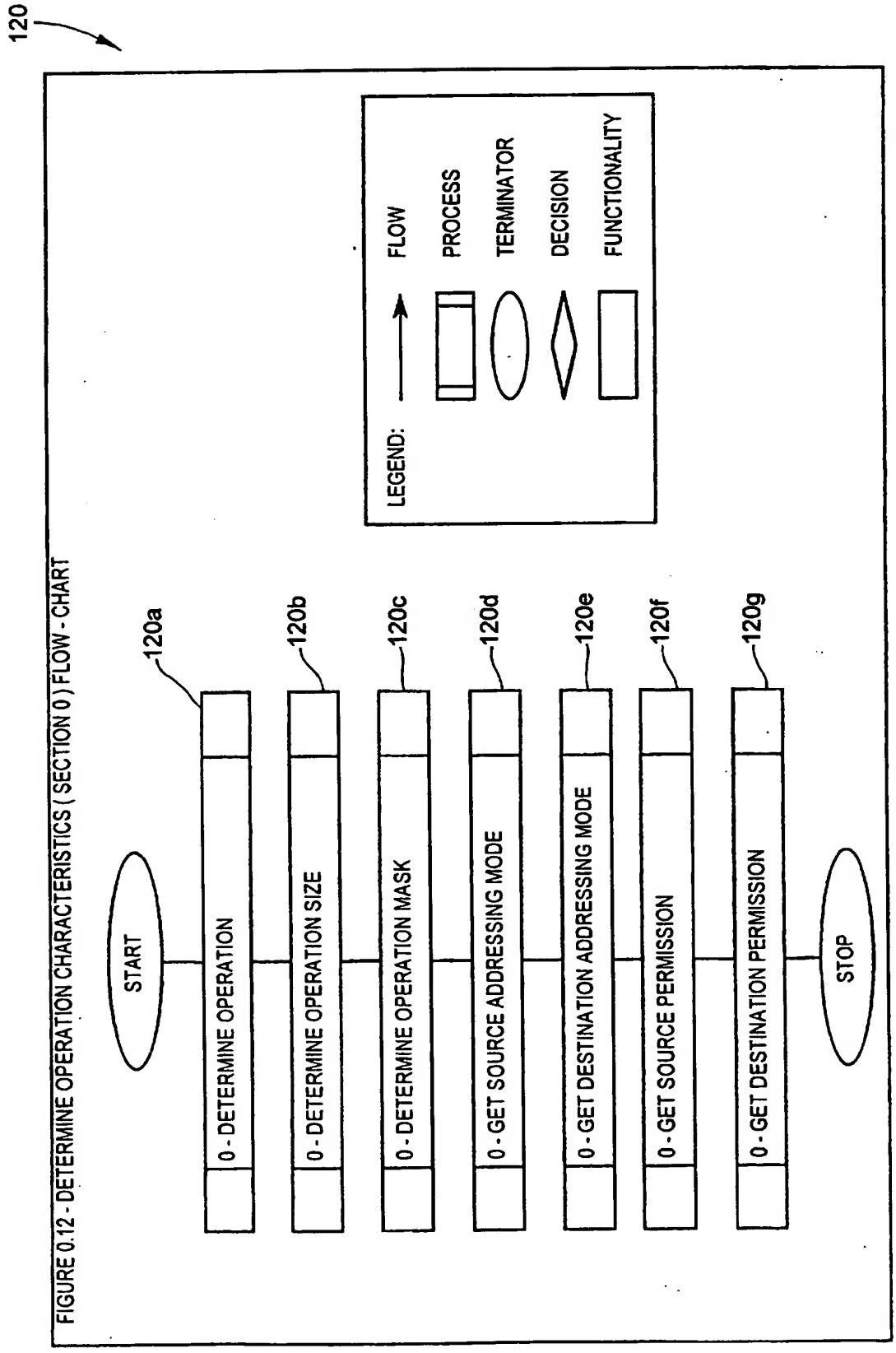

120
FIGURE 0.12 - DETERMINE OPERATION CHARACTERISTICS (SECTION 0) FLOW - CHART
START
0 - DETERMINE OPERATION
0 - DETERMINE OPERATION SIZE
0 - DETERMINE OPERATION MASK
0 - GET SOURCE ADDRESSING MODE
0 - GET DESTINATION ADDRESSING MODE
0 - GET SOURCE PERMISSION
0 - GET DESTINATION PERMISSION
STOP
120a
120b
120c
120d
120e
120f
120g
LEGEND:
FLOW
PROCESS
TERMINATOR
DECISION
FUNCTIONALITY

FIG. 14

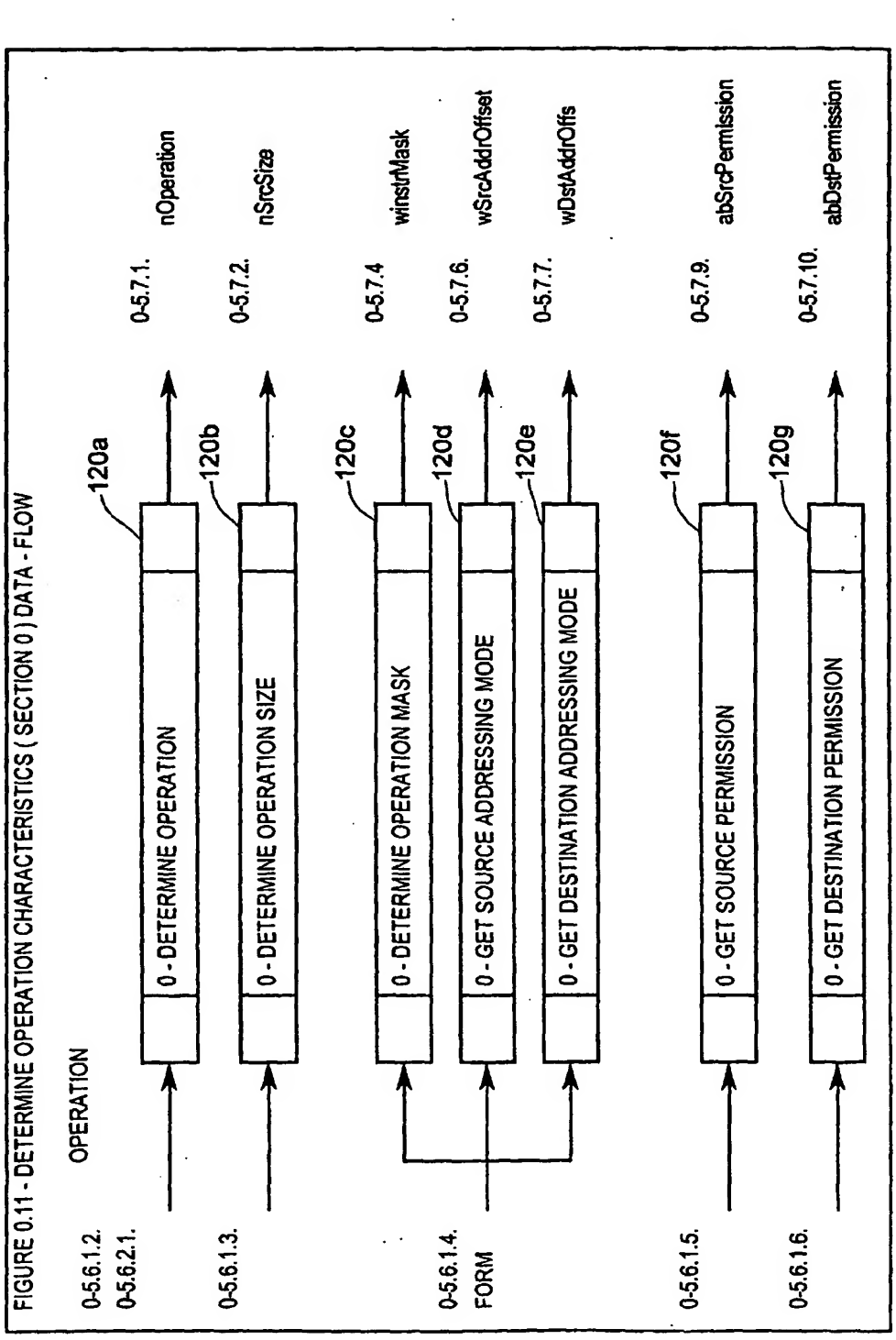

120
FIGURE 0.11 - DETERMINE OPERATION CHARACTERISTICS (SECTION 0) DATA - FLOW
OPERATION
0-5.6.1.2.
0-5.6.2.1.
0-5.6.1.3.
0-5.6.1.4.
FORM
0-5.6.1.5.
0-5.6.1.6.
120a
120b
120c
120d
120e
120f
120g
0 - DETERMINE OPERATION
0 - DETERMINE OPERATION SIZE
0 - DETERMINE OPERATION MASK
0 - GET SOURCE ADDRESSING MODE
0 - GET DESTINATION ADDRESSING MODE
0 - GET SOURCE PERMISSION
0 - GET DESTINATION PERMISSION
0-5.7.1.
nOperation
0-5.7.2.
nSrcSize
0-5.7.4
winstrMask
0-5.7.6.
wSrcAddrOffset
0-5.7.7.
wDstAddrOffs
0-5.7.9.
abSrcPermission
0-5.7.10.
abDstPermission

FIG. 15

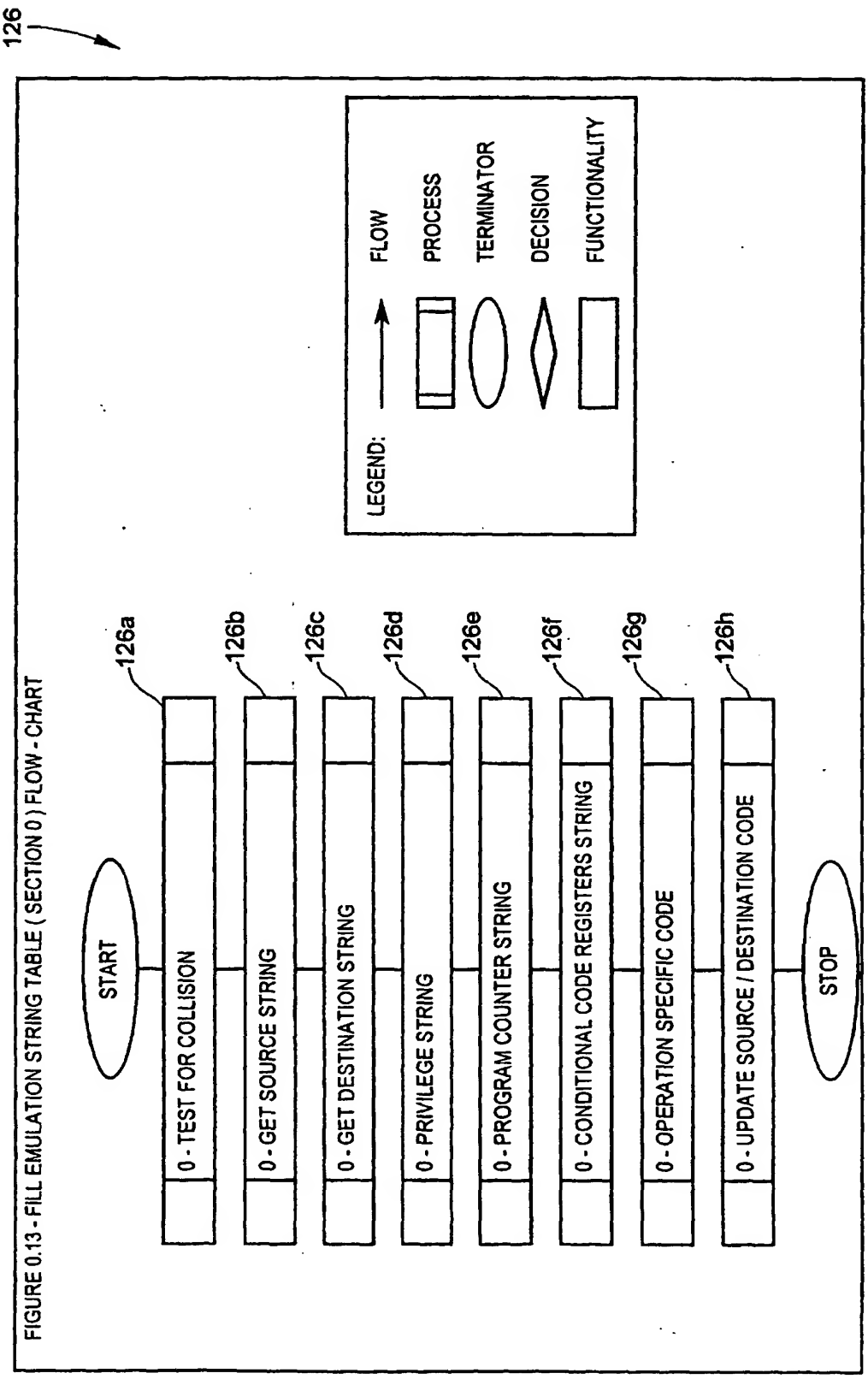

126
FIGURE 0.13 - FILL EMULATION STRING TABLE (SECTION 0) FLOW - CHART
START
0 - TEST FOR COLLISION
126a
0 - GET SOURCE STRING
126b
0 - GET DESTINATION STRING
126c
0 - PRIVILEGE STRING
126d
0 - PROGRAM COUNTER STRING
126e
0 - CONDITIONAL CODE REGISTERS STRING
126f
0 - OPERATION SPECIFIC CODE
126g
0 - UPDATE SOURCE / DESTINATION CODE
126h
STOP
LEGEND:
FLOW
PROCESS
TERMINATOR
DECISION
FUNCTIONALITY

FIG. 16

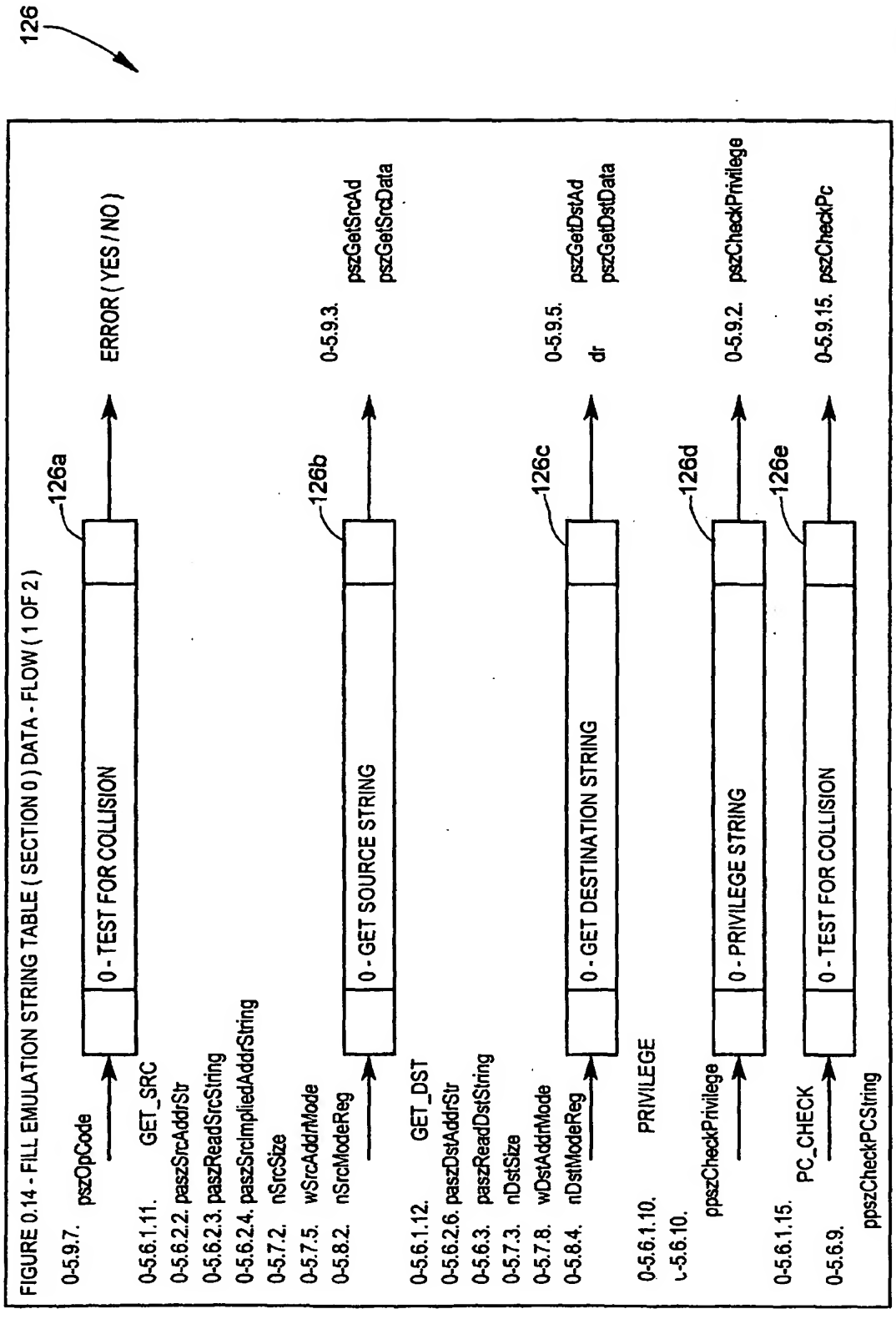

126
FIGURE 0.14 - FILL EMULATION STRING TABLE (SECTION 0) DATA - FLOW (1 OF 2)
0-5.9.7. pszOpCode
126a
0 - TEST FOR COLLISION
ERROR (YES / NO)
0-5.6.1.11. GET_SRC
0-5.6.2.2. paszSrcAddrStr
0-5.6.2.3. paszReadSrcString
0-5.6.2.4. paszSrcImpliedAddrString
0-5.7.2. nSrcSize
0-5.7.5. wSrcAddrMode
0-5.8.2. nSrcModeReg
126b
0 - GET SOURCE STRING
0-5.9.3. pszGetSrcAd
pszGetSrcData
0-5.6.1.12. GET_DST
0-5.6.2.6. paszDstAddrStr
0-5.6.3. paszReadDstString
0-5.7.3. nDstSize
0-5.7.8. wDstAddrMode
0-5.8.4. nDstModeReg
126c
0 - GET DESTINATION STRING
0-5.9.5. pszGetDstAd
dr pszGetDstData
0-5.6.1.10. PRIVILEGE
ι-5.6.10.
ppszCheckPrivilege
126d
0 - PRIVILEGE STRING
0-5.9.2. pszCheckPrivilege
0-5.6.1.15. PC_CHECK
0-5.6.9. ppszCheckPCString
126e
0 - TEST FOR COLLISION
0-5.9.15. pszCheckPc

FIG. 17

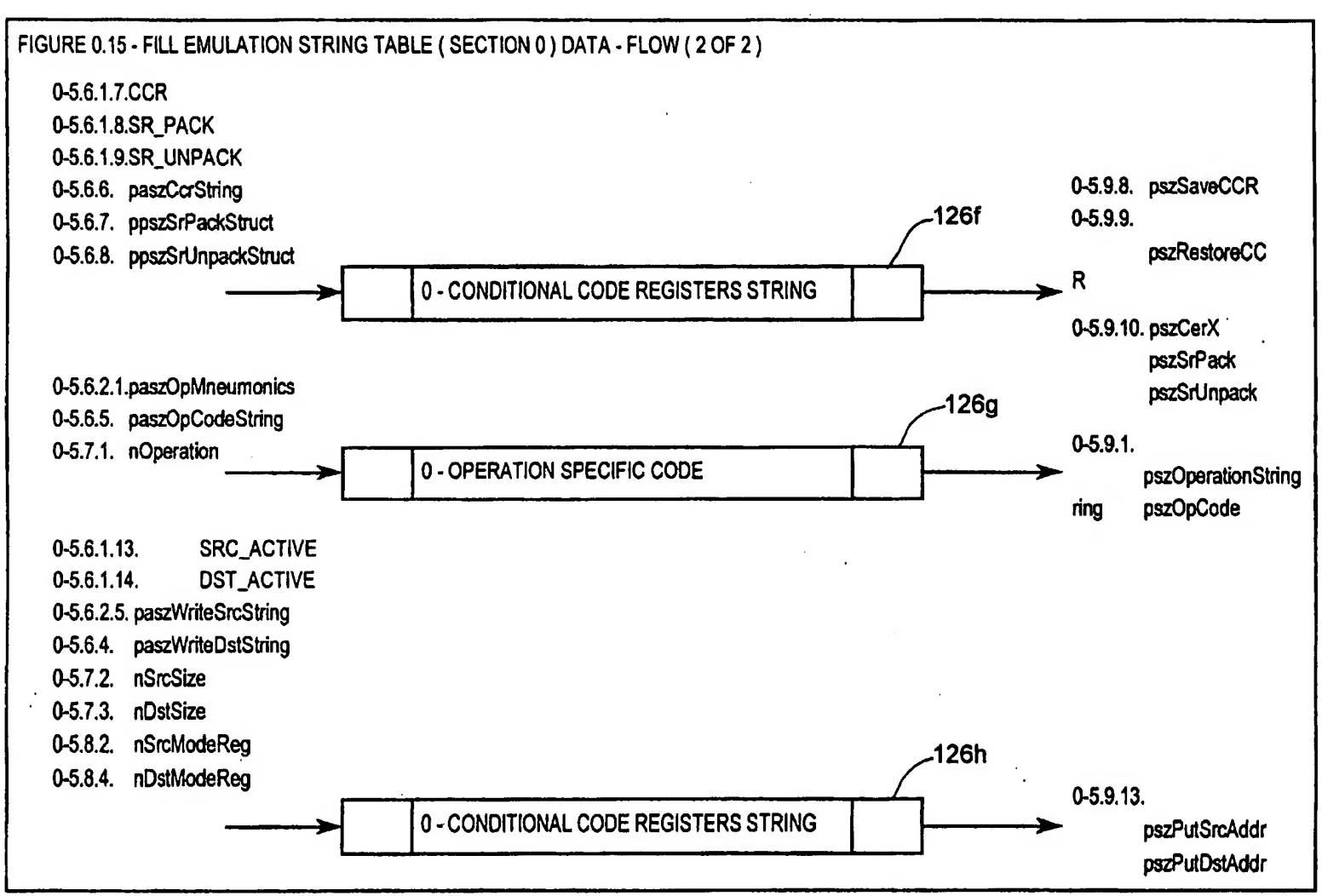

126
FIGURE 0.15 - FILL EMULATION STRING TABLE (SECTION 0) DATA - FLOW (2 OF 2)
0-5.6.1.7.CCR
0-5.6.1.8.SR_PACK
0-5.6.1.9.SR_UNPACK
0-5.6.6. paszCcrString
0-5.6.7. ppszSrPackStruct
0-5.6.8. ppszSrUnpackStruct
126f
0 - CONDITIONAL CODE REGISTERS STRING
0-5.9.8. pszSaveCCR
0-5.9.9. pszRestoreCCR
0-5.9.10. pszCerX
pszSrPack
pszSrUnpack
0-5.6.2.1.paszOpMneumonics
0-5.6.5. paszOpCodeString
0-5.7.1. nOperation
126g
0 - OPERATION SPECIFIC CODE
0-5.9.1. pszOperationString
pszOpCode
0-5.6.1.13. SRC_ACTIVE
0-5.6.1.14. DST_ACTIVE
0-5.6.2.5. paszWriteSrcString
0-5.6.4. paszWriteDstString
0-5.7.2. nSrcSize
0-5.7.3. nDstSize
0-5.8.2. nSrcModeReg
0-5.8.4. nDstModeReg
126h
0 - CONDITIONAL CODE REGISTERS STRING
0-5.9.13. pszPutSrcAddr
pszPutDstAddr

FIG. 18

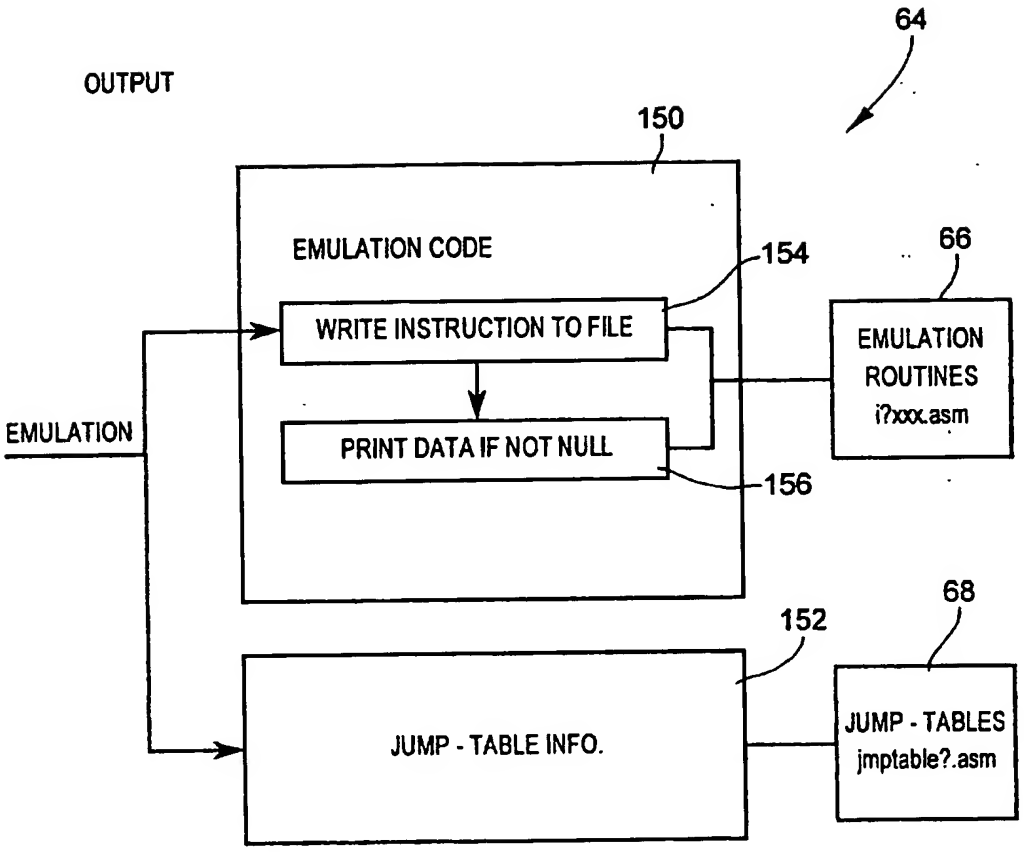

64
OUTPUT
150
EMULATION CODE
154
66
WRITE INSTRUCTION TO FILE
EMULATION ROUTINES i?xxx.asm
EMULATION
PRINT DATA IF NOT NULL
156
68
152
JUMP - TABLE INFO.
JUMP - TABLES jmptable?.asm

FIG.19

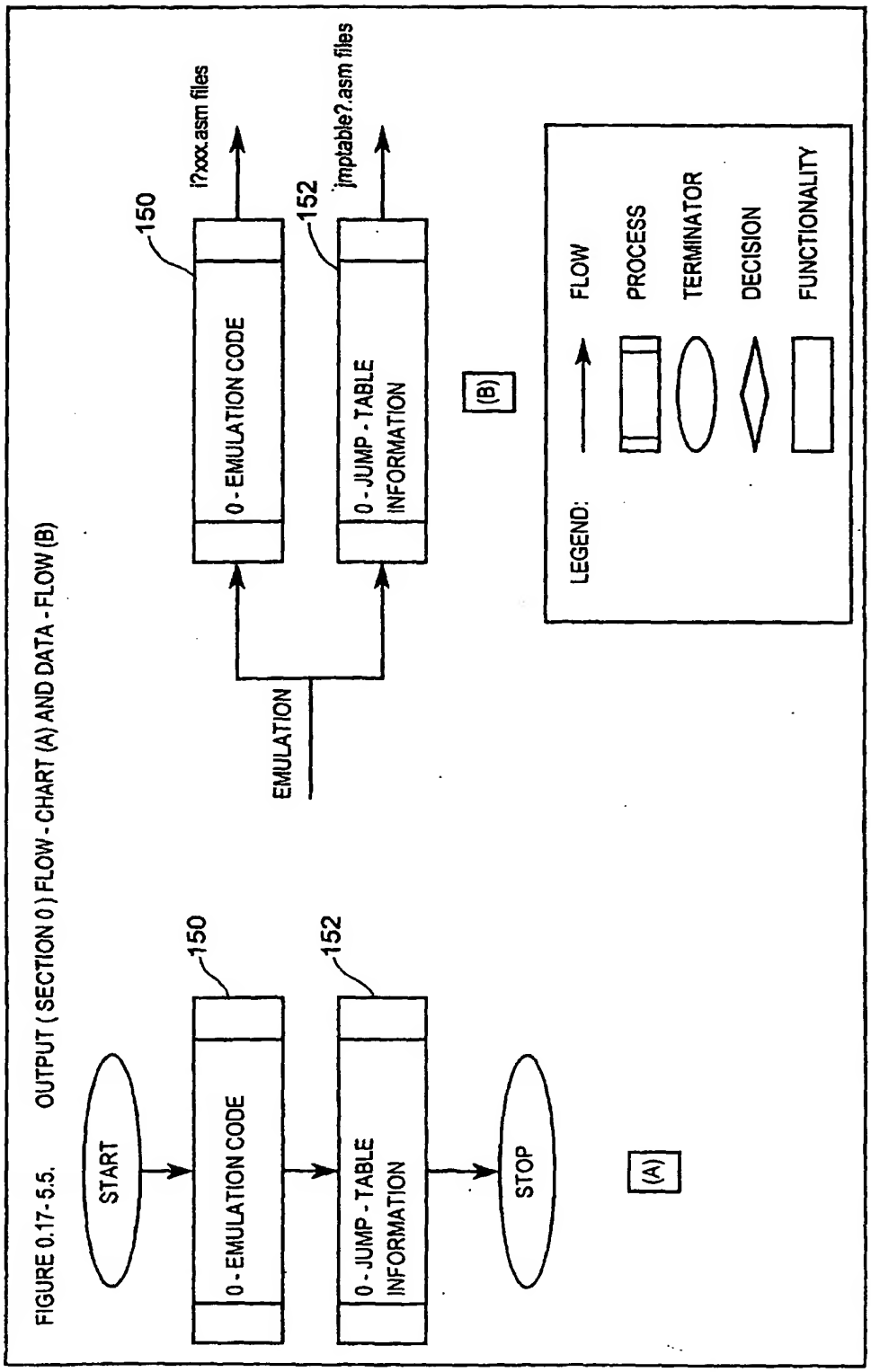


FIGURE 0.17-5.5. OUTPUT (SECTION 0) FLOW - CHART (A) AND DATA - FLOW (B)
START
0 - EMULATION CODE
150
0 - JUMP - TABLE INFORMATION
152
STOP
(A)
EMULATION
0 - EMULATION CODE
150
i?xxx.asm files
0 - JUMP - TABLE INFORMATION
152
jmptable?.asm files
(B)
LEGEND:
FLOW
PROCESS
TERMINATOR
DECISION
FUNCTIONALITY

FIG. 20

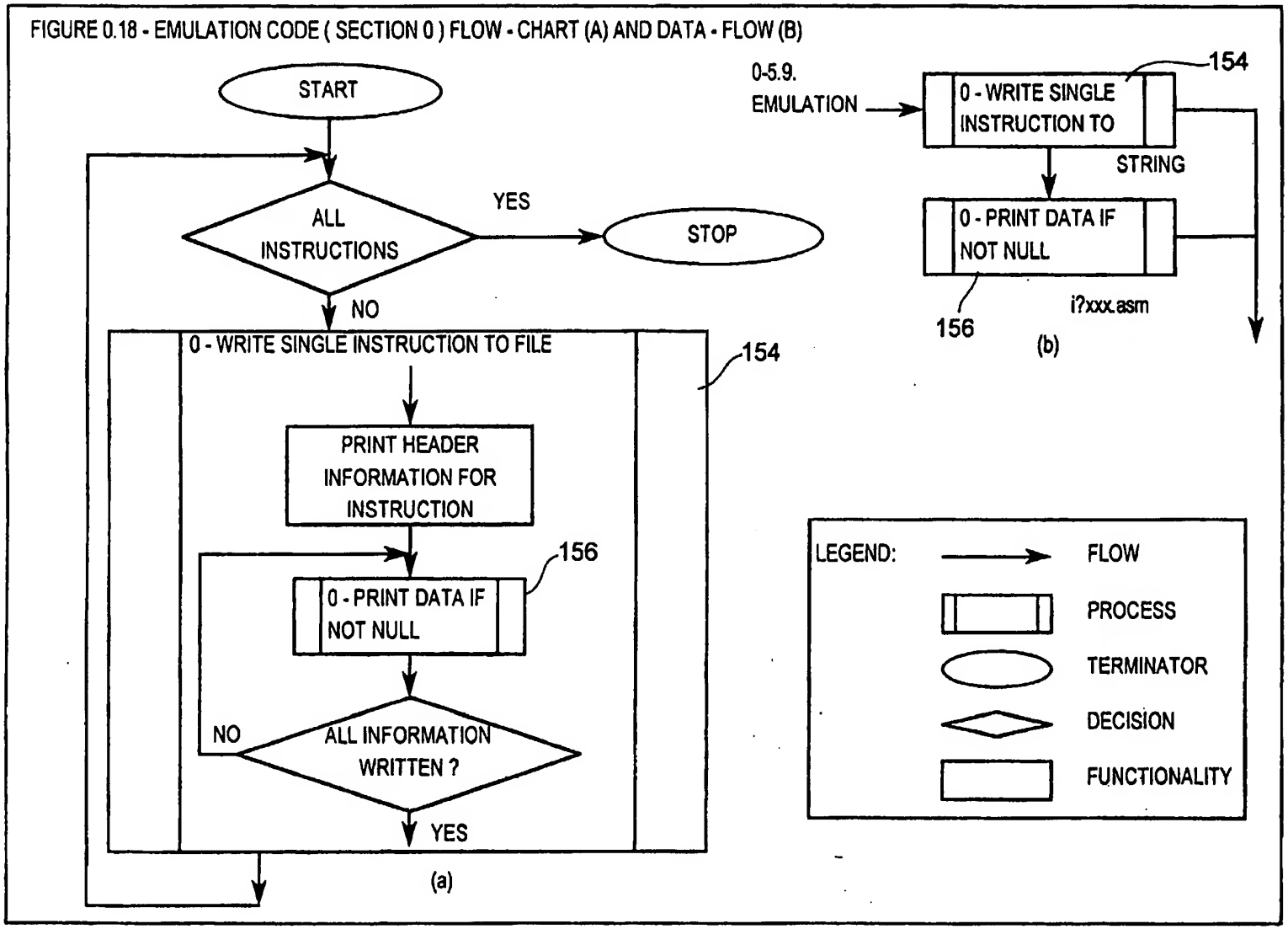


FIGURE 0.18 - EMULATION CODE (SECTION 0) FLOW - CHART (A) AND DATA - FLOW (B)
START
ALL INSTRUCTIONS
YES
STOP
NO
0 - WRITE SINGLE INSTRUCTION TO FILE
154
PRINT HEADER INFORMATION FOR INSTRUCTION
156
0 - PRINT DATA IF NOT NULL
NO
ALL INFORMATION WRITTEN ?
YES
(a)
0-5.9. EMULATION
0 - WRITE SINGLE INSTRUCTION TO STRING
154
0 - PRINT DATA IF NOT NULL
156
i?xxx.asm
(b)
LEGEND:
FLOW
PROCESS
TERMINATOR
DECISION
FUNCTIONALITY

FIG. 21

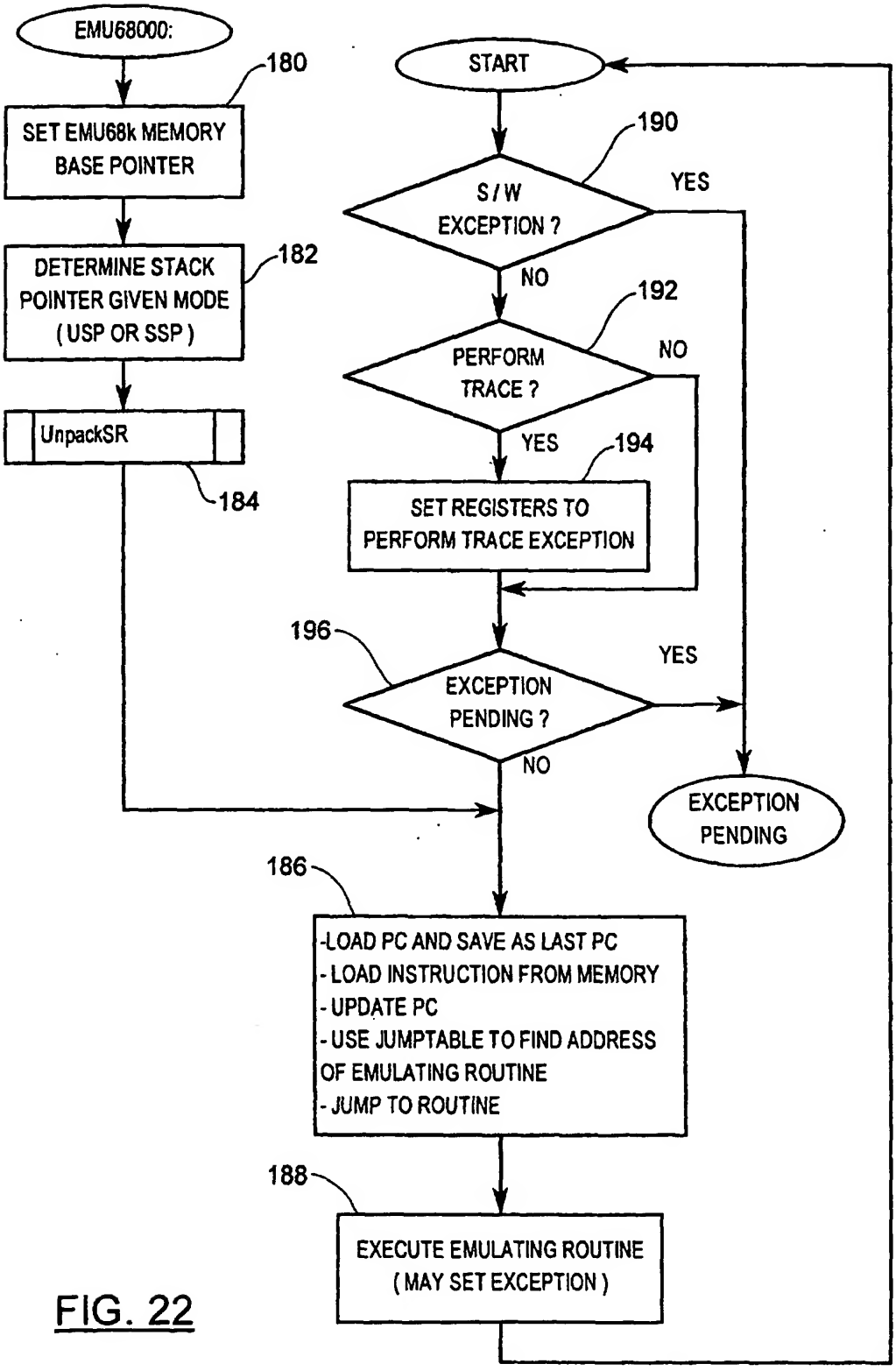

EMU68000:
180
SET EMU68k MEMORY BASE POINTER
DETERMINE STACK POINTER GIVEN MODE (USP OR SSP)
182
UnpackSR
184
START
190
S / W EXCEPTION ?
YES
NO
192
PERFORM TRACE ?
NO
YES
194
SET REGISTERS TO PERFORM TRACE EXCEPTION
196
EXCEPTION PENDING ?
YES
NO
EXCEPTION PENDING
186
-LOAD PC AND SAVE AS LAST PC
- LOAD INSTRUCTION FROM MEMORY
- UPDATE PC
- USE JUMPTABLE TO FIND ADDRESS OF EMULATING ROUTINE
- JUMP TO ROUTINE
188
EXECUTE EMULATING ROUTINE (MAY SET EXCEPTION)

FIG. 22

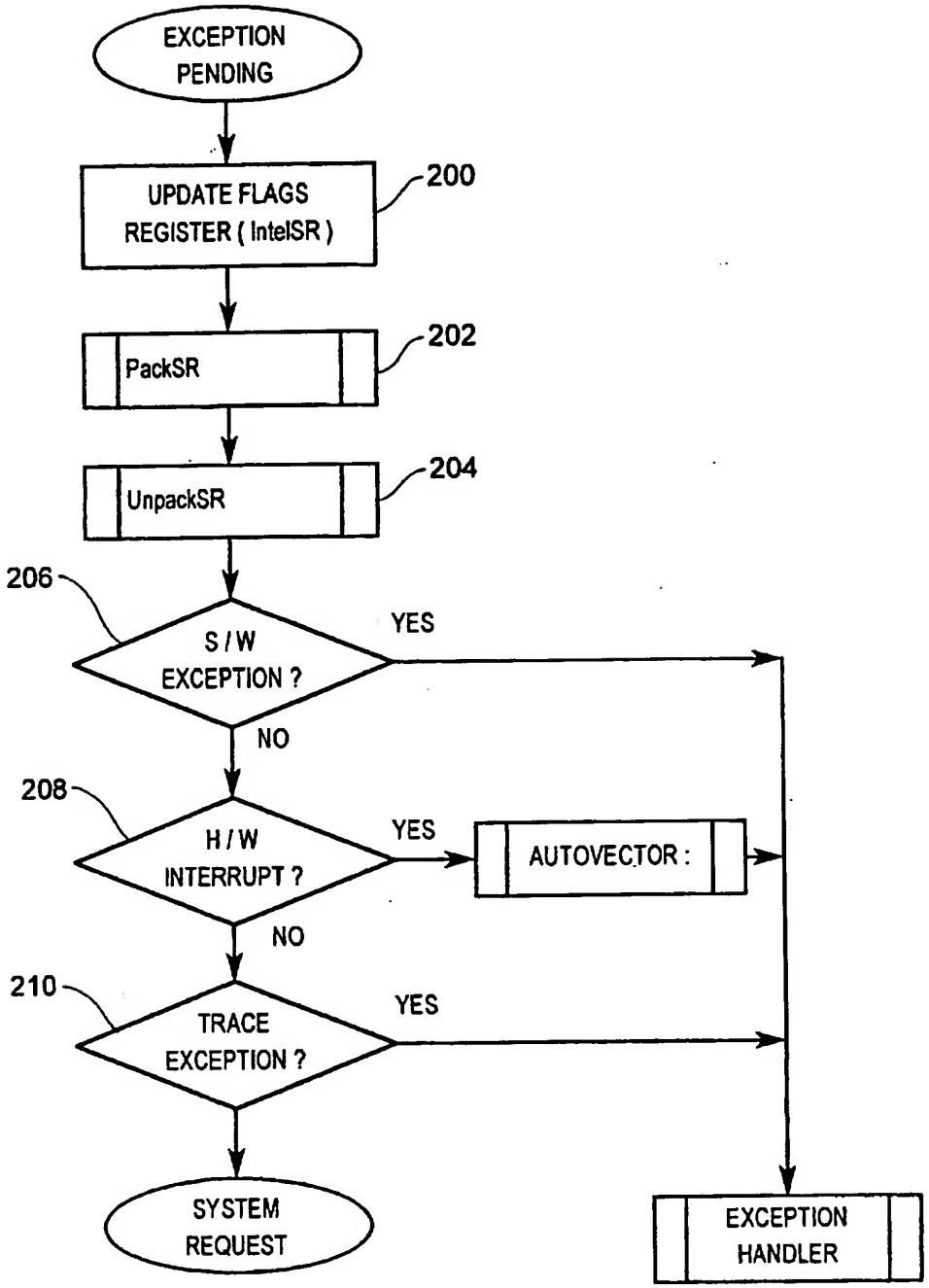

EXCEPTION PENDING
UPDATE FLAGS REGISTER (IntelSR)
200
PackSR
202
UnpackSR
204
206
S / W EXCEPTION ?
YES
NO
208
H / W INTERRUPT ?
YES
AUTOVECTOR :
NO
210
TRACE EXCEPTION ?
YES
SYSTEM REQUEST
EXCEPTION HANDLER

FIG. 23

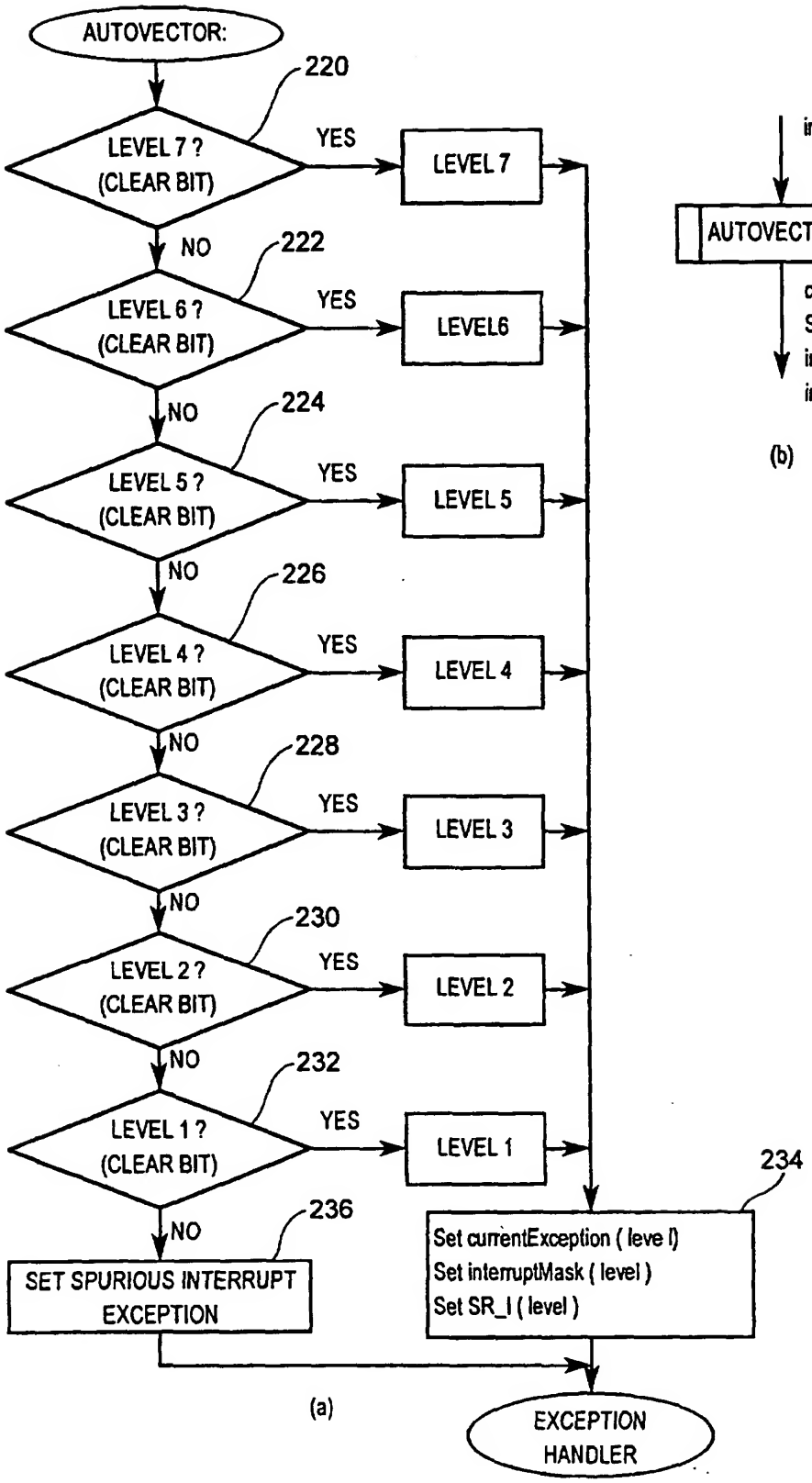

AUTOVECTOR:
220
LEVEL 7 ?
(CLEAR BIT)
YES
LEVEL 7
NO
222
LEVEL 6 ?
(CLEAR BIT)
YES
LEVEL6
NO
224
LEVEL 5 ?
(CLEAR BIT)
YES
LEVEL 5
NO
226
LEVEL 4 ?
(CLEAR BIT)
YES
LEVEL 4
NO
228
LEVEL 3 ?
(CLEAR BIT)
YES
LEVEL 3
NO
230
LEVEL 2 ?
(CLEAR BIT)
YES
LEVEL 2
NO
232
LEVEL 1 ?
(CLEAR BIT)
YES
LEVEL 1
NO
236
SET SPURIOUS INTERRUPT EXCEPTION
234
Set currentException (level)
Set interruptMask (level)
Set SR_I (level)
EXCEPTION HANDLER
(a)
interruptRequest
AUTOVECTOR
currentException
SR_I
interruptMask
interruptRequest
(b)

FIG. 24

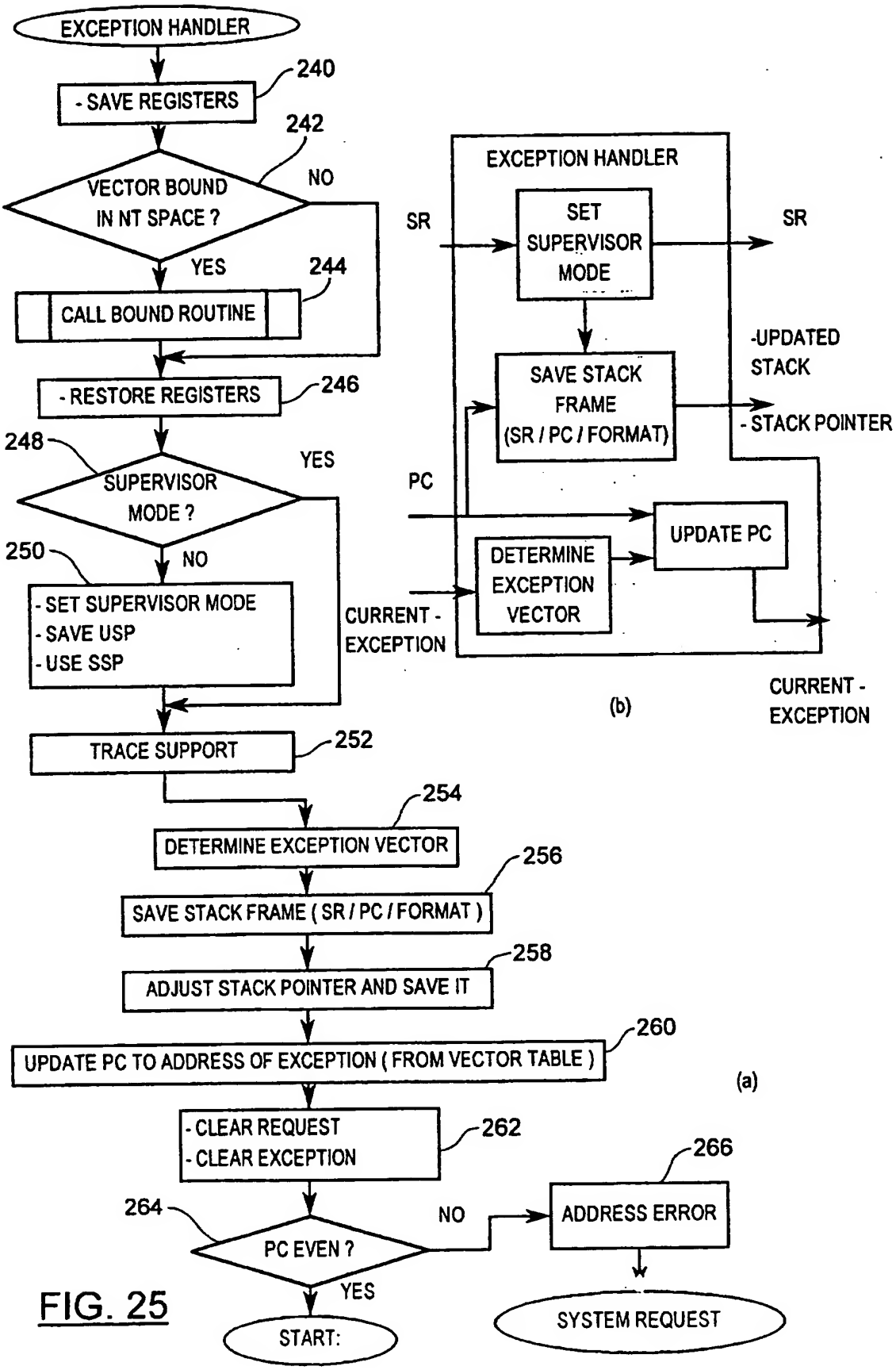

EXCEPTION HANDLER
240
- SAVE REGISTERS
242
VECTOR BOUND IN NT SPACE ?
NO
YES
244
CALL BOUND ROUTINE
- RESTORE REGISTERS
246
248
SUPERVISOR MODE ?
YES
NO
250
- SET SUPERVISOR MODE
- SAVE USP
- USE SSP
TRACE SUPPORT
252
254
DETERMINE EXCEPTION VECTOR
256
SAVE STACK FRAME (SR / PC / FORMAT)
258
ADJUST STACK POINTER AND SAVE IT
260
UPDATE PC TO ADDRESS OF EXCEPTION (FROM VECTOR TABLE)
- CLEAR REQUEST
- CLEAR EXCEPTION
262
264
PC EVEN ?
NO
YES
START:
266
ADDRESS ERROR
SYSTEM REQUEST
(a)
EXCEPTION HANDLER
SR
SET SUPERVISOR MODE
SR
-UPDATED STACK
SAVE STACK FRAME (SR / PC / FORMAT)
- STACK POINTER
PC
UPDATE PC
DETERMINE EXCEPTION VECTOR
CURRENT - EXCEPTION
CURRENT - EXCEPTION
(b)

FIG. 25

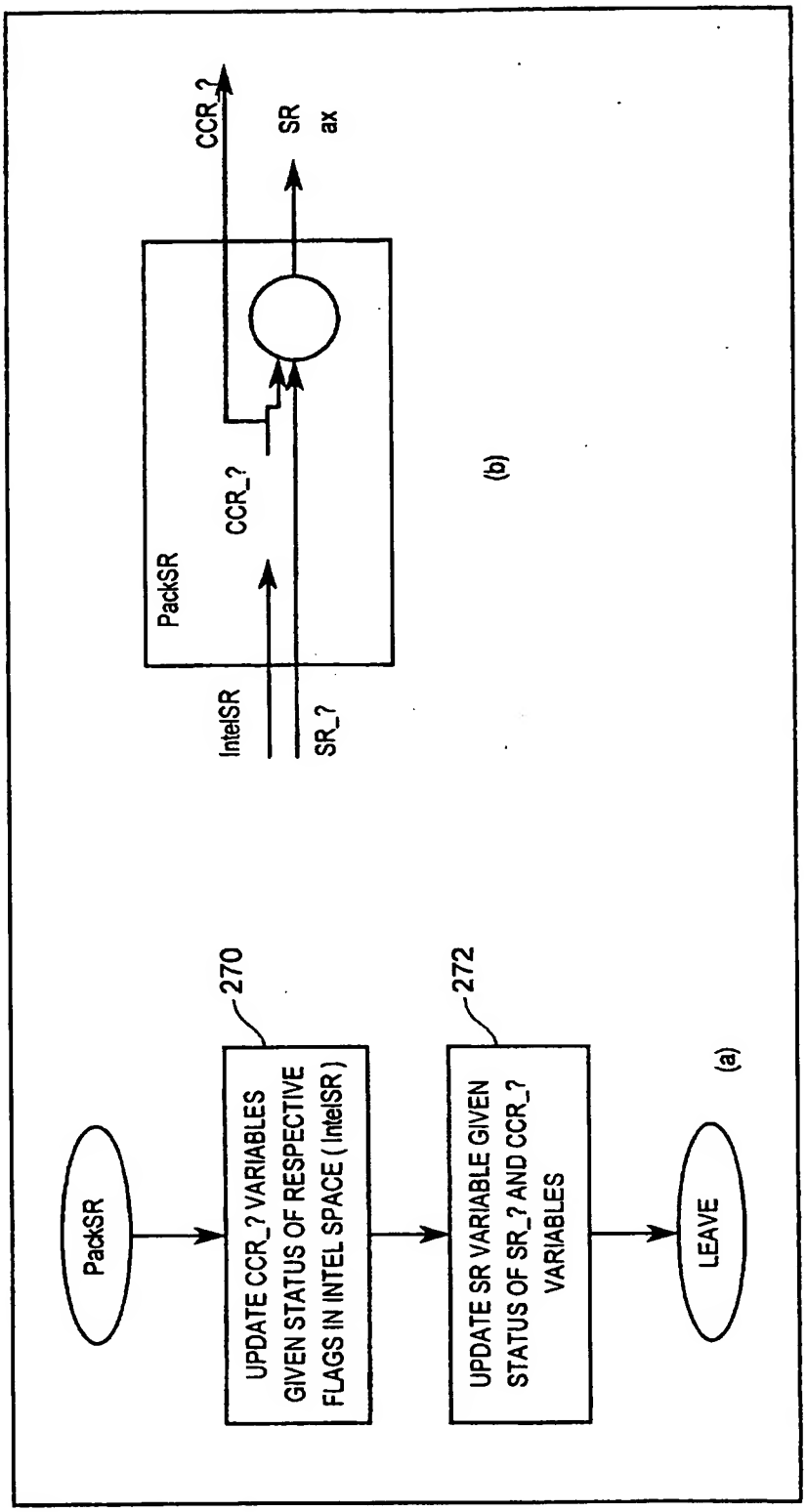

PackSR
270
UPDATE CCR_? VARIABLES GIVEN STATUS OF RESPECTIVE FLAGS IN INTEL SPACE (IntelSR)
272
UPDATE SR VARIABLE GIVEN STATUS OF SR_? AND CCR_? VARIABLES
LEAVE
(a)
PackSR
CCR_?
IntelSR
SR_?
CCR_?
SR
ax
(b)

FIG. 26

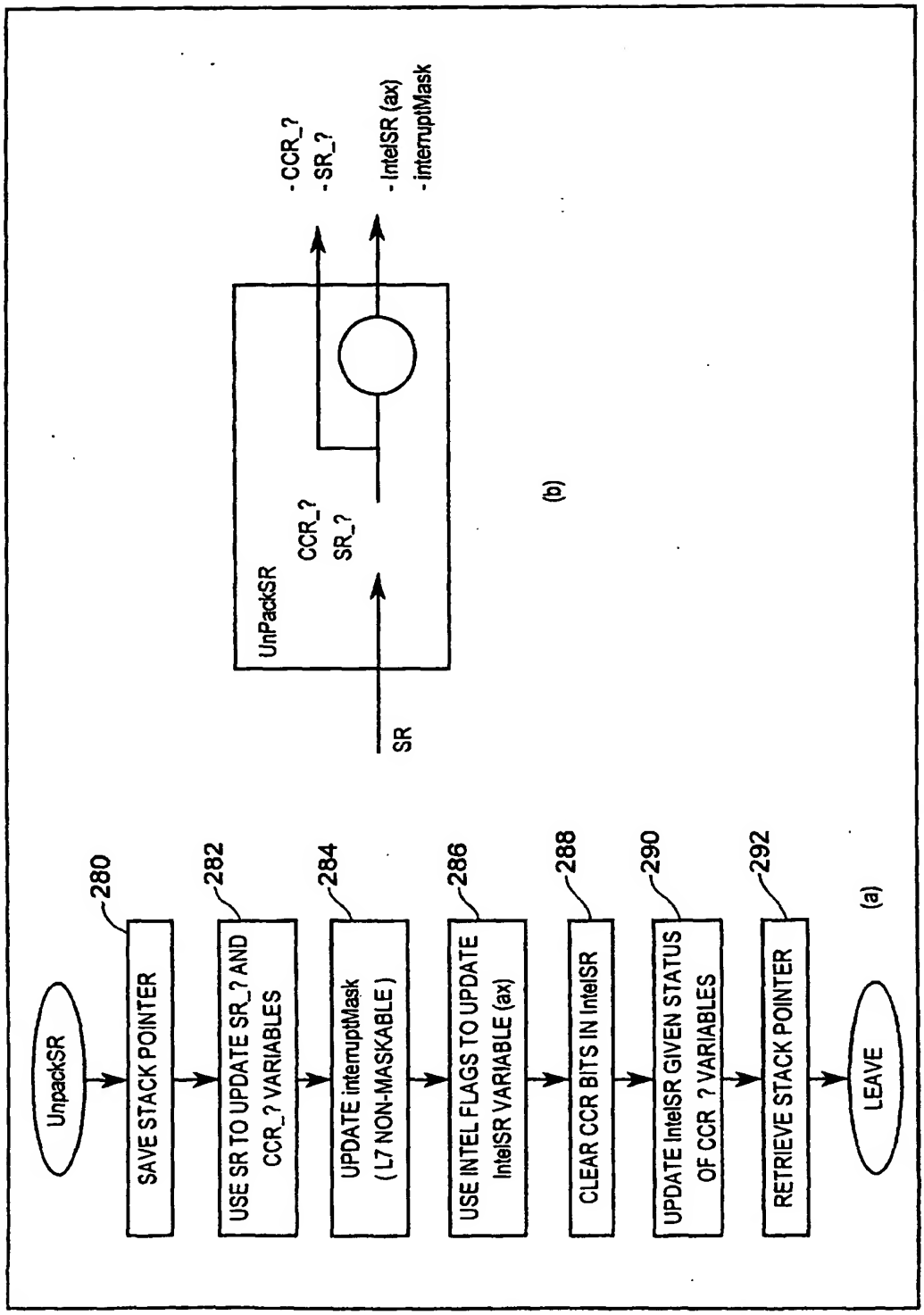

UnpackSR
280
SAVE STACK POINTER
282
USE SR TO UPDATE SR_? AND CCR_? VARIABLES
284
UPDATE interruptMask (L7 NON-MASKABLE)
286
USE INTEL FLAGS TO UPDATE IntelSR VARIABLE (ax)
288
CLEAR CCR BITS IN IntelSR
290
UPDATE IntelSR GIVEN STATUS OF CCR_? VARIABLES
292
RETRIEVE STACK POINTER
LEAVE
(a)
UnPackSR
CCR_?
SR_?
SR
-CCR_?
-SR_?
-IntelSR (ax)
-interruptMask
(b)

FIG. 27

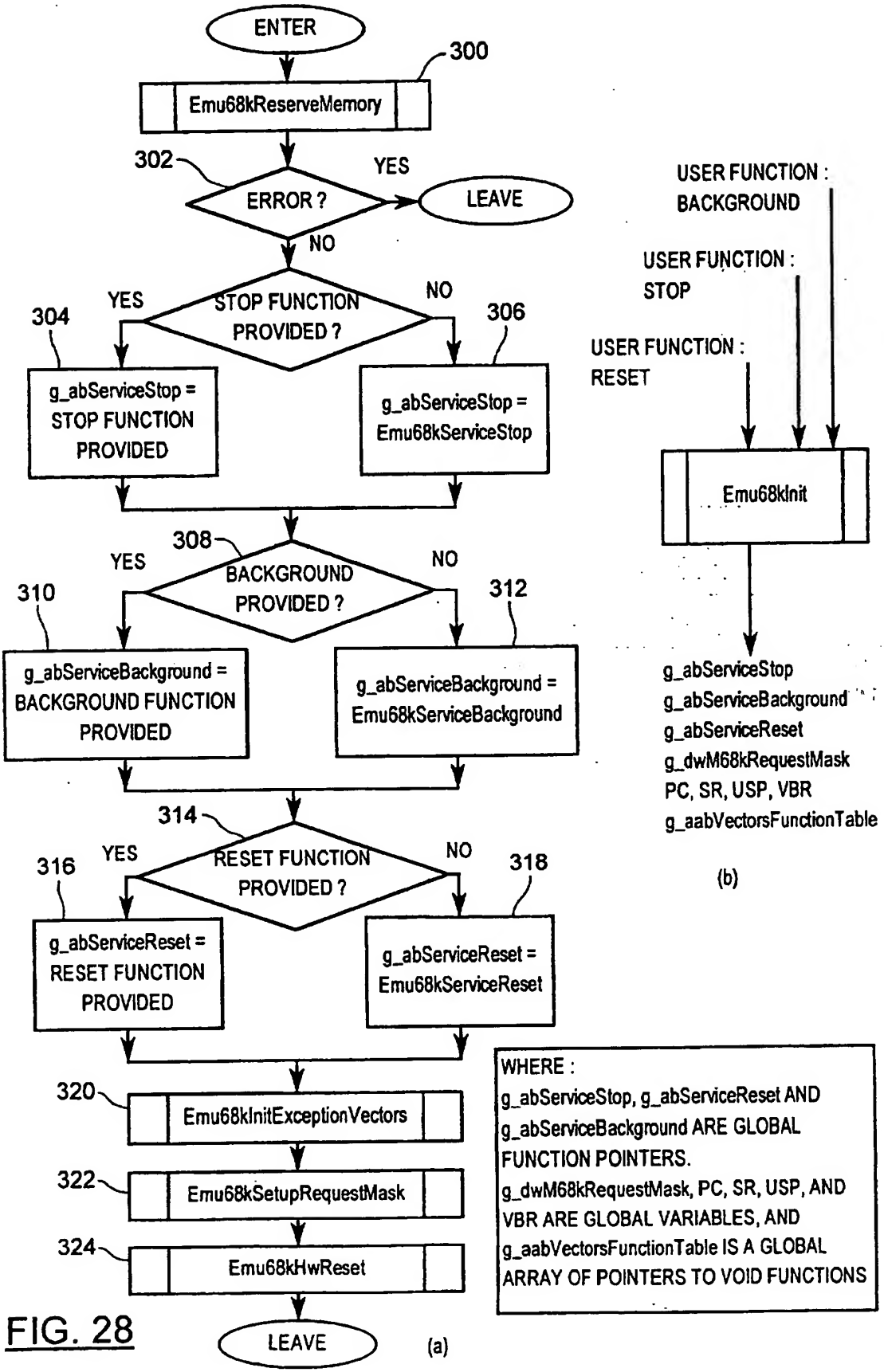

ENTER
300
Emu68kReserveMemory
302
ERROR ?
YES
LEAVE
NO
STOP FUNCTION PROVIDED ?
YES
NO
304
306
g_abServiceStop = STOP FUNCTION PROVIDED
g_abServiceStop = Emu68kServiceStop
308
BACKGROUND PROVIDED ?
YES
NO
310
312
g_abServiceBackground = BACKGROUND FUNCTION PROVIDED
g_abServiceBackground = Emu68kServiceBackground
314
RESET FUNCTION PROVIDED ?
YES
NO
316
318
g_abServiceReset = RESET FUNCTION PROVIDED
g_abServiceReset = Emu68kServiceReset
320
Emu68kInitExceptionVectors
322
Emu68kSetupRequestMask
324
Emu68kHwReset
LEAVE
(a)
USER FUNCTION : BACKGROUND
USER FUNCTION : STOP
USER FUNCTION : RESET
Emu68kInit
g_abServiceStop
g_abServiceBackground
g_abServiceReset
g_dwM68kRequestMask
PC, SR, USP, VBR
g_aabVectorsFunctionTable
(b)
WHERE :
g_abServiceStop, g_abServiceReset AND g_abServiceBackground ARE GLOBAL FUNCTION POINTERS.
g_dwM68kRequestMask, PC, SR, USP, AND VBR ARE GLOBAL VARIABLES, AND
g_aabVectorsFunctionTable IS A GLOBAL ARRAY OF POINTERS TO VOID FUNCTIONS

FIG. 28

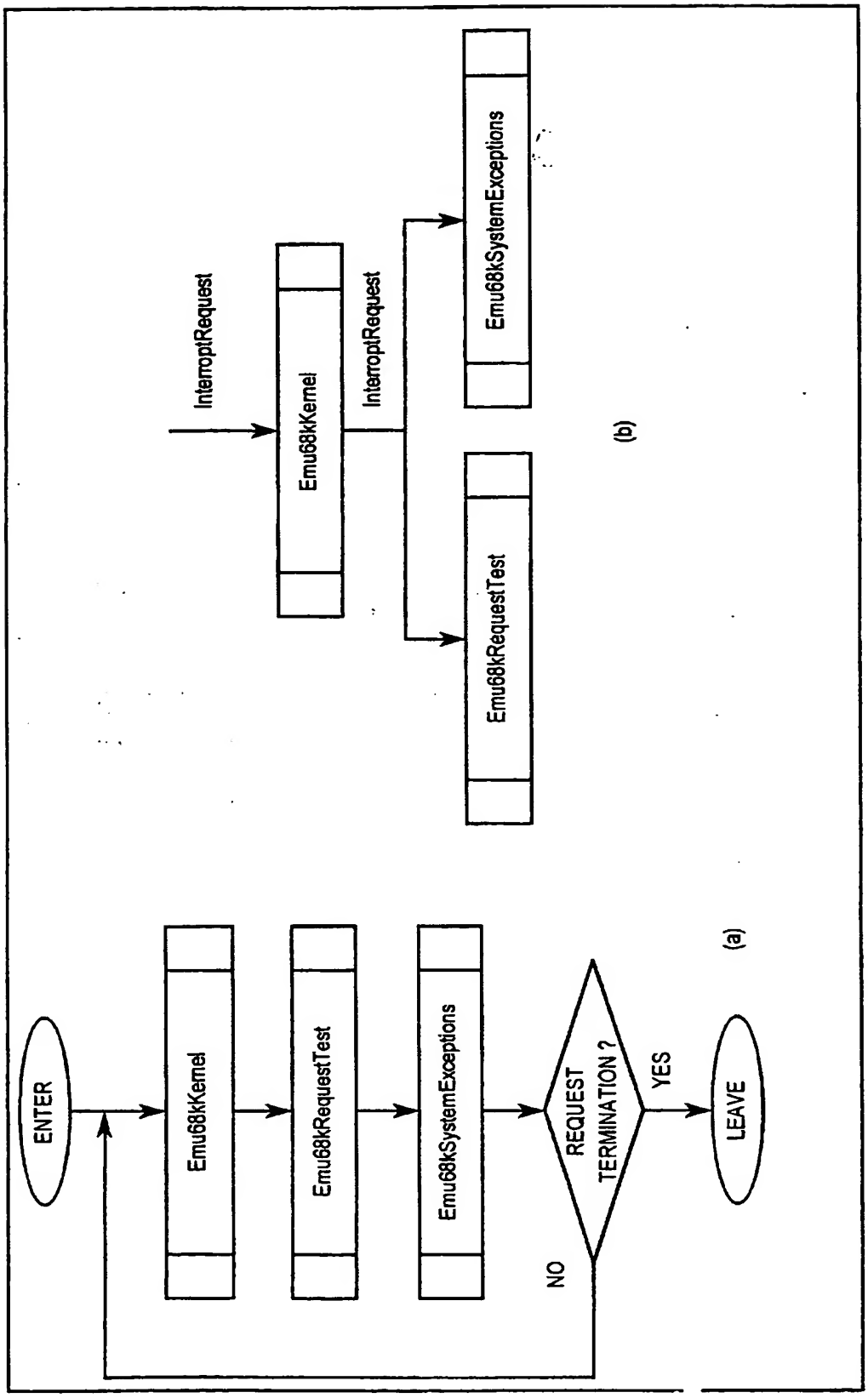

ENTER
Emu68kKernel
Emu68kRequestTest
Emu68kSystemExceptions
REQUEST TERMINATION ?
NO
YES
LEAVE
(a)
InterruptRequest
Emu68kKernel
InterruptRequest
Emu68kSystemExceptions
Emu68kRequestTest
(b)

FIG. 29

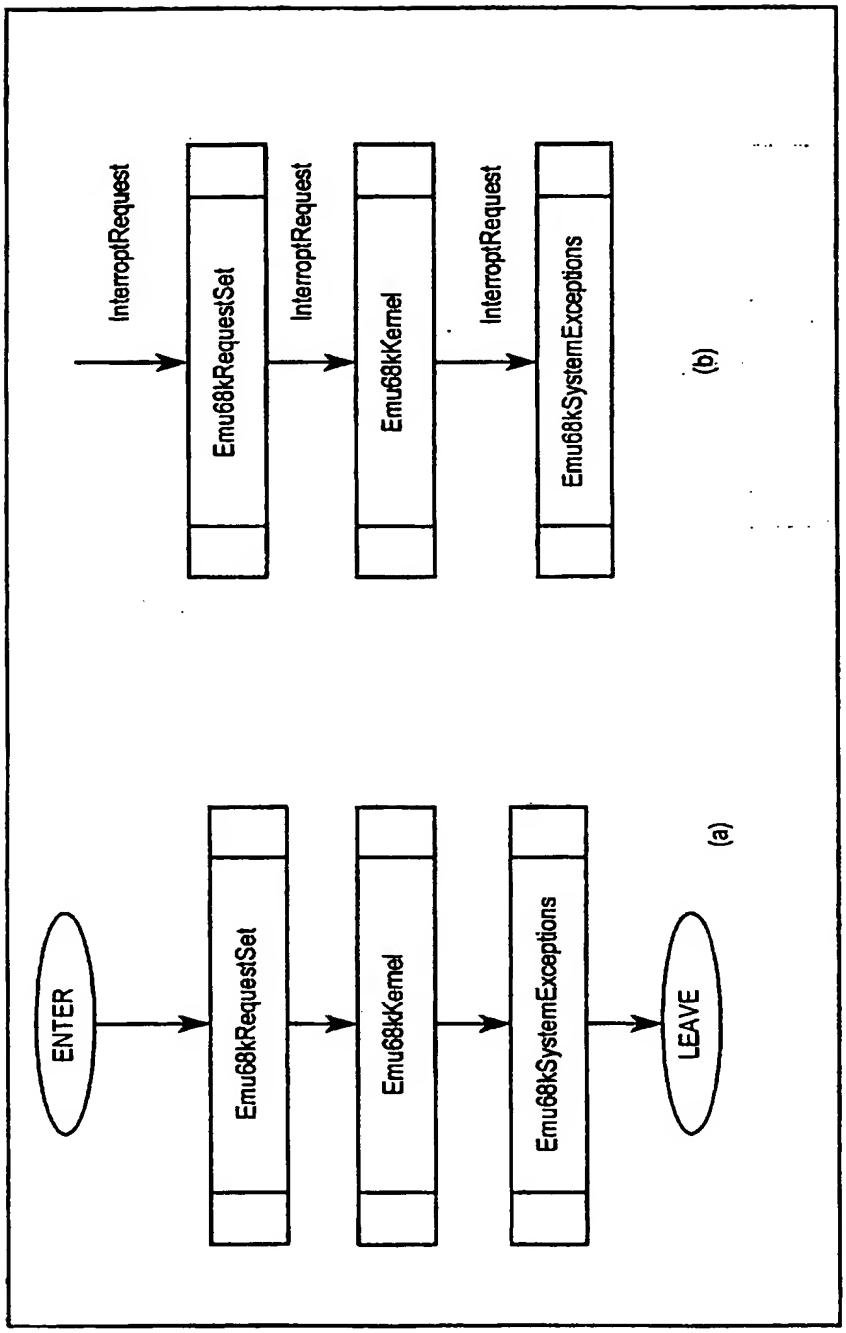

ENTER
Emu68kRequestSet
Emu68kKernel
Emu68kSystemExceptions
LEAVE
(a)
InterroptRequest
Emu68kRequestSet
InterroptRequest
Emu68kKernel
InterroptRequest
Emu68kSystemExceptions
(b)

FIG. 30

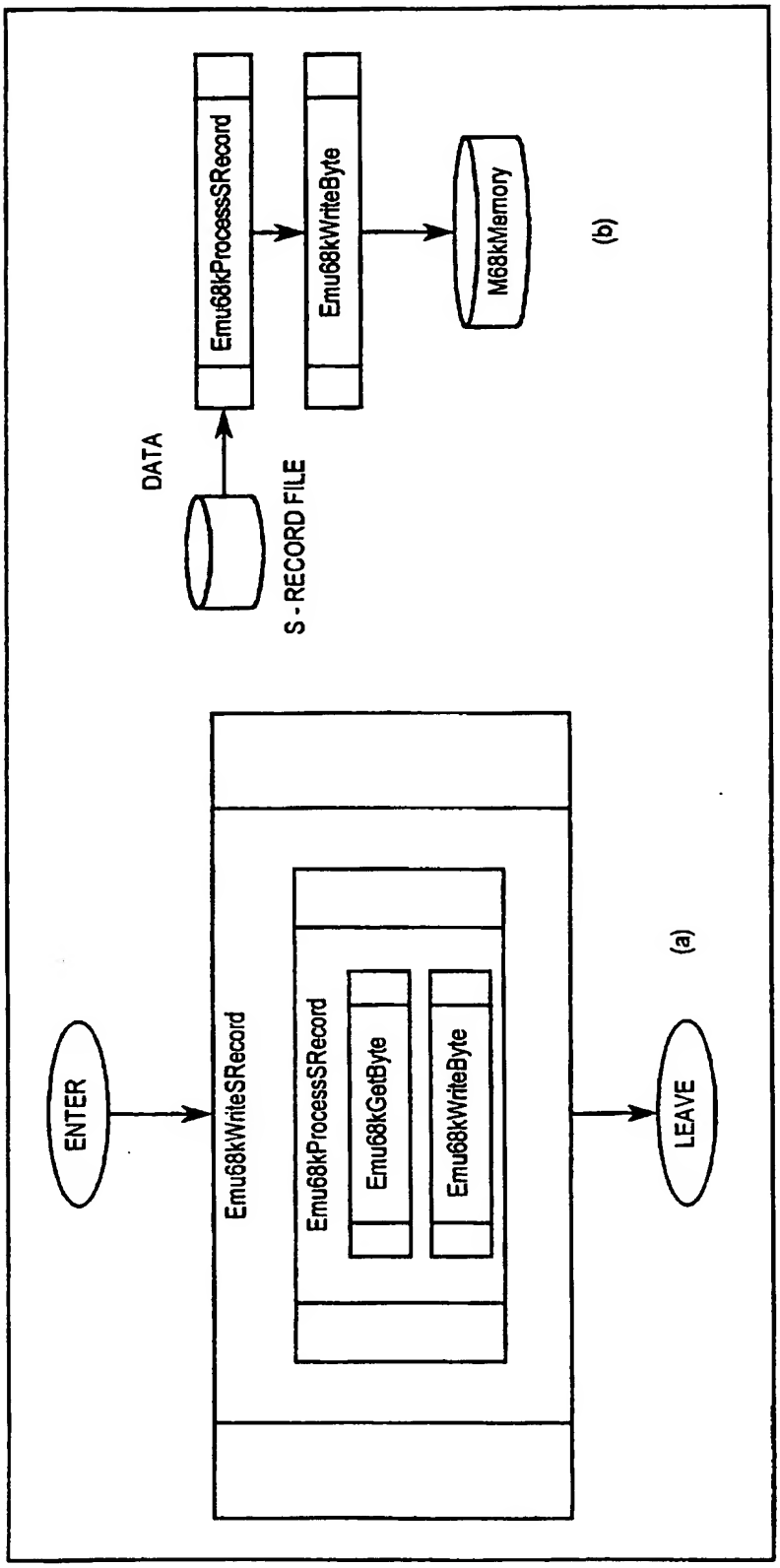

ENTER
Emu68kWriteSRecord
Emu68kProcessSRecord
Emu68kGetByte
Emu68kWriteByte
LEAVE
(a)
DATA
S - RECORD FILE
Emu68kProcessSRecord
Emu68kWriteByte
M68kMemory
(b)

FIG. 31

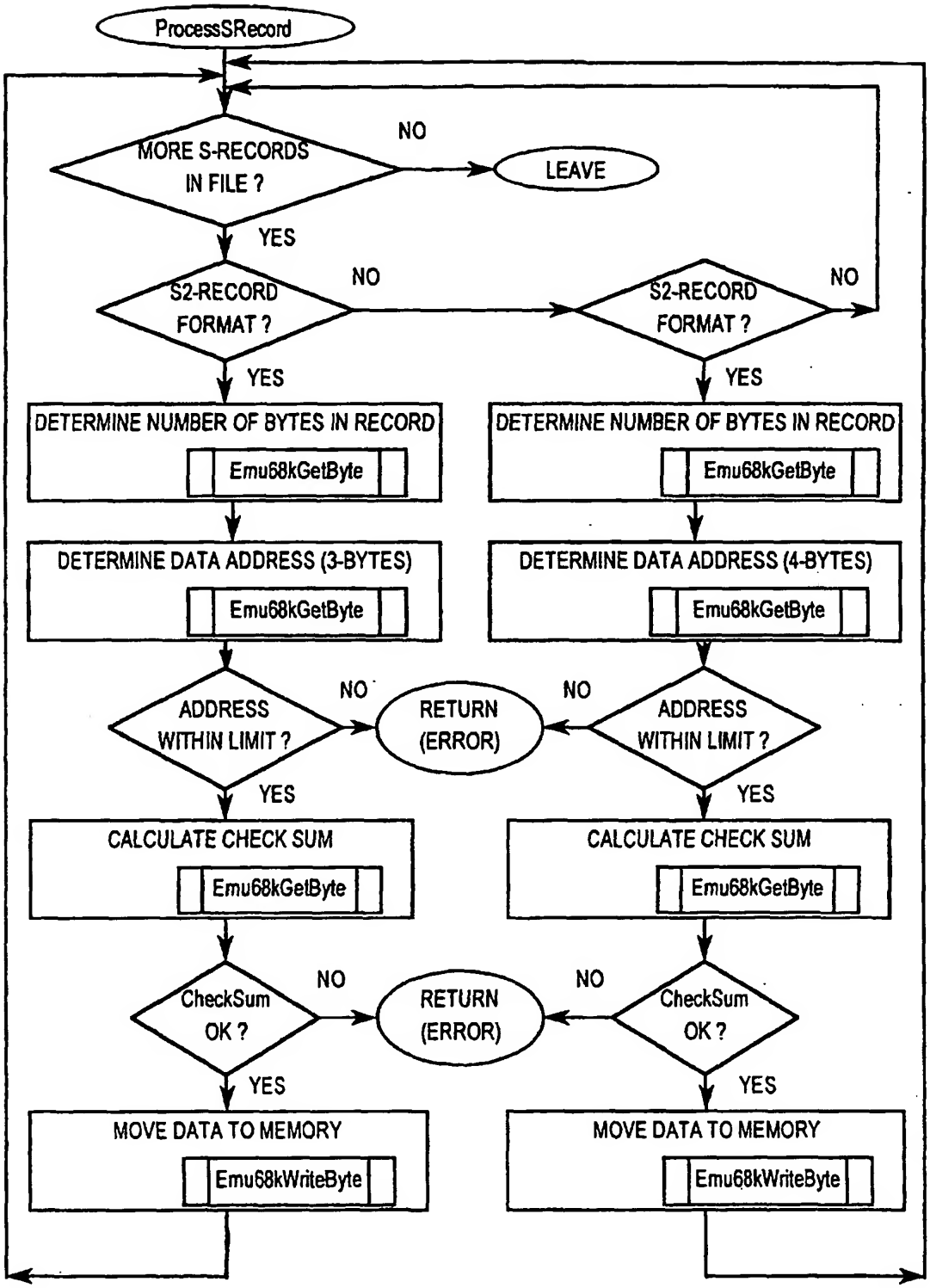

ProcessSRecord
MORE S-RECORDS IN FILE ?
NO
LEAVE
YES
S2-RECORD FORMAT ?
NO
S2-RECORD FORMAT ?
NO
YES
YES
DETERMINE NUMBER OF BYTES IN RECORD
Emu68kGetByte
DETERMINE NUMBER OF BYTES IN RECORD
Emu68kGetByte
DETERMINE DATA ADDRESS (3-BYTES)
Emu68kGetByte
DETERMINE DATA ADDRESS (4-BYTES)
Emu68kGetByte
ADDRESS WITHIN LIMIT ?
NO
RETURN (ERROR)
NO
ADDRESS WITHIN LIMIT ?
YES
YES
CALCULATE CHECK SUM
Emu68kGetByte
CALCULATE CHECK SUM
Emu68kGetByte
CheckSum OK ?
NO
RETURN (ERROR)
NO
CheckSum OK ?
YES
YES
MOVE DATA TO MEMORY
Emu68kWriteByte
MOVE DATA TO MEMORY
Emu68kWriteByte

FIG. 32

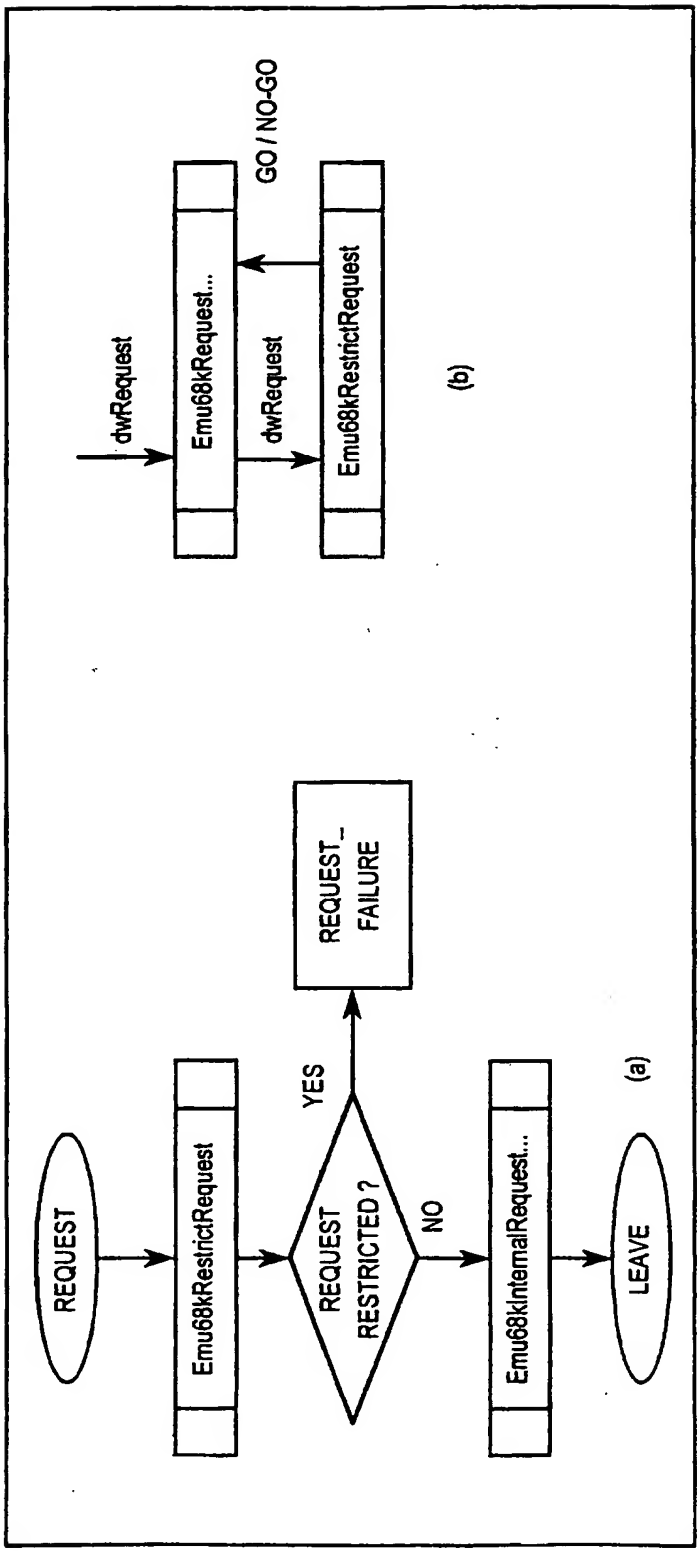

REQUEST
Emu68kRestrictRequest
REQUEST RESTRICTED ?
YES
REQUEST_ FAILURE
NO
Emu68kInternalRequest...
LEAVE
(a)
dwRequest
Emu68kRequest...
dwRequest
Emu68kRestrictRequest
GO / NO-GO
(b)

FIG. 33

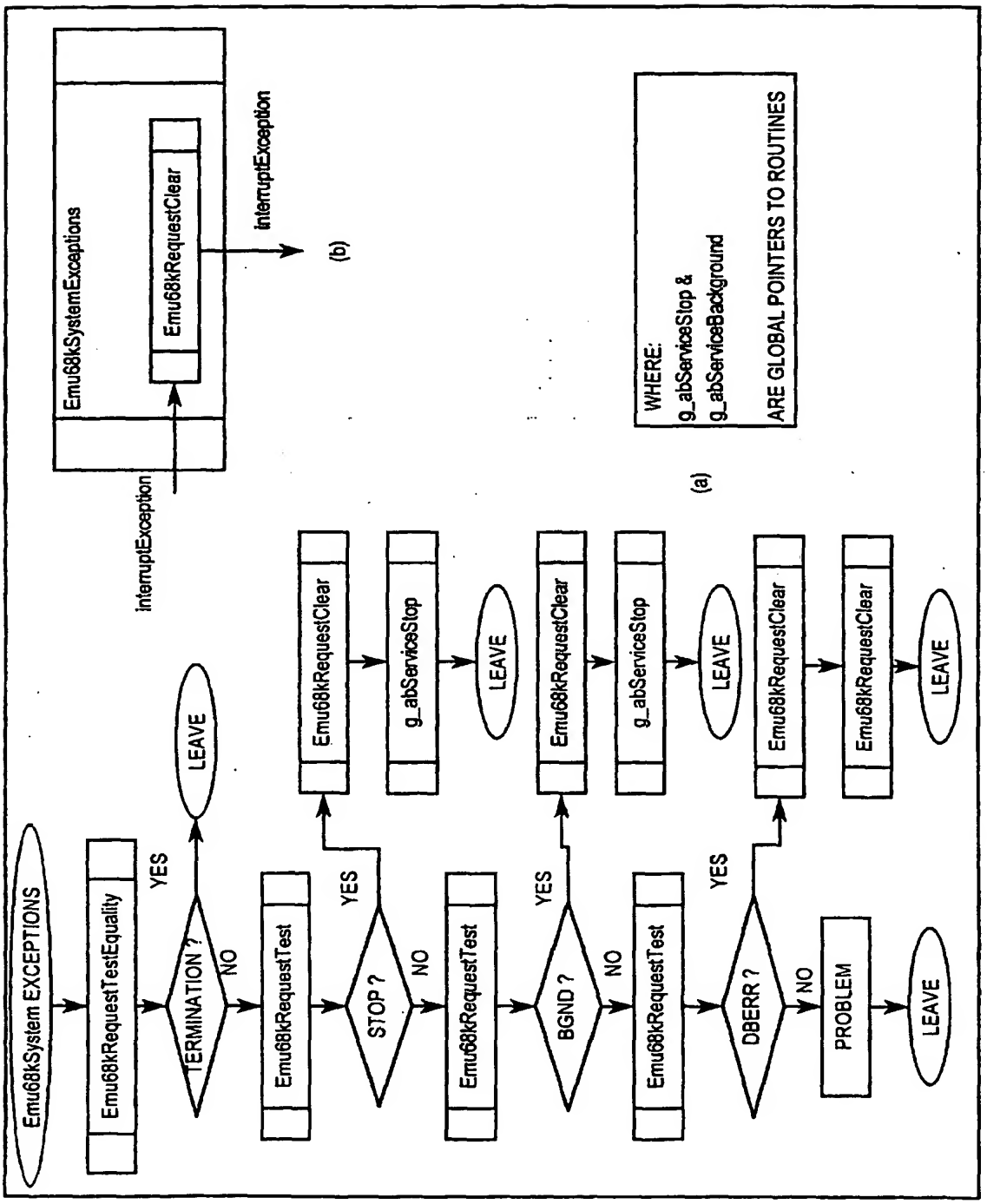

Emu68kSystem EXCEPTIONS
Emu68kRequestTestEquality
TERMINATION ?
YES
NO
LEAVE
Emu68kRequestTest
STOP ?
YES
NO
Emu68kRequestClear
g_abServiceStop
LEAVE
Emu68kRequestTest
BGND ?
YES
NO
Emu68kRequestClear
g_abServiceStop
LEAVE
Emu68kRequestTest
DBERR ?
YES
NO
Emu68kRequestClear
Emu68kRequestClear
LEAVE
PROBLEM
LEAVE
interruptException
Emu68kSystemExceptions
Emu68kRequestClear
interruptException
(b)
(a)
WHERE:
g_abServiceStop &
g_abServiceBackground
ARE GLOBAL POINTERS TO ROUTINES

FIG. 34

EMULATION SYSTEM INCLUDING EMULATOR GENERATOR

Field Of The Invention

The present invention relates generally to microprocessor platforms and in particular to an emulation system including an emulator generator to permit code developed for a specific computing environment to be executed on microprocessor platforms of different design.

Background Of The Invention

A variety of microprocessor platforms exist on the market today. However, new microprocessor platforms are being introduced with increasing frequency. As new microprocessor platforms emerge and old microprocessor platforms become obsolete, it is desirable to retain the use of legacy code (i.e. code written for older obsolete microprocessor platforms) due to the investment of time and expertise legacy code represents. In order to retain the use and functionality of legacy code in new microprocessor platforms, the legacy code must either be ported, re-written or emulated.

Unfortunately, porting and re-writing of legacy code requires a substantial investment of time and effort. Emulation however, provides an opportunity to use legacy code on new microprocessor platforms thereby allowing new tools, applications and/or functionality to be accessed while minimizing time and/or resource commitments.

Although emulation provides advantages over porting and re-writing, software emulation in the past has been generally too slow to provide sufficient performance to be feasible and thus, has seldom been considered.

It is therefore an object of the present invention to provide an emulation system to permit code developed for a specific computing environment to be executed on microprocessor platforms of different design which obviates or mitigates the disadvantages associated with the prior art.

Summary Of The Invention

According to one aspect of the present invention there is provided an emulation system to allow code written for a specific computing environment to be run on a different microprocessor platform comprising:

an emulation routine generator including a table storing code segments corresponding to instructions of said specific computing environment, said emulation routine generator creating said emulation routines written in code for said different microprocessor platform from selected code segments in said table, said emulation routines corresponding to instructions of said specific computing environment;

an emulator kernel to read instructions of said specific computing environment and access and execute emulation routines thereby to emulate and carry out said read instructions; and

an interface resembling said specific computing environment.

A link between the instruction (16-bit pattern for a Motorola 68000) and the emulating routine is contained within a jump-table prepared by the generator.

Brief Description Of The Drawings

An embodiment of the present invention will now be described more fully with reference to the accompanying drawings in which:

Figure 1 is a schematic block diagram of an emulation system in accordance with the present invention;

Figure 2 is a schematic block diagram of an emulator generator forming part of the emulation system of Figure 1;

Figure 3 is a schematic block diagram of an M68k emulator kernel forming part of the emulation system of Figure 1;

Figure 4 is a schematic block diagram of an emulation API forming part of the emulation system of Figure 1;

Figure 5 is schematic block diagram of an input component forming part of the emulator generator of Figure 2;

Figure 6 shows a flow chart and a data flow diagram illustrating the overall operation of the input component of Figure 5;

Figure 7 shows a flow chart illustrating an initialize static strings function performed by the input component of Figure 5;

Figure 8 shows a data flow diagram illustrating the initialize static strings function of Figure 7;

Figure 9 shows a flow chart illustrating a read operations format function performed by the input component of Figure 5;

Figure 10 shows a data flow diagram illustrating the read operations format function of Figure 9;

Figure 11 is a schematic block diagram of an emulation routine generation component forming part of the emulator generator of Figure 2;

Figure 12 shows a flow chart illustrating the overall operation of the emulation routine generation component of Figure 11;

Figure 13 shows a data flow diagram of Figure 12;

Figure 14 shows a flow chart illustrating a determine operations characteristics function performed by the emulation routine generation component of Figure 11;

Figure 15 shows a data flow diagram illustrating the determine operation characteristics function of Figure 14;

Figure 16 shows a flow chart illustrating a fill emulation string table function performed by the emulation routine generation component of Figure 11;

Figures 17 and 18 show a data flow diagram illustrating the fill emulation string table function of Figure 16;

Figure 19 is a schematic block diagram of an output component forming part of the emulator generator of Figure 2;

Figure 20 shows a flow chart and a data flow diagram illustrating the overall operation of the output component of Figure 19;

Figure 21 shows a flow chart and a data flow diagram illustrating an emulation code function performed by the output component of Figure 19;

Figure 22 shows a flow chart illustrating an emulation loop executed by the M68k emulator kernel of Figure 3;

Figure 23 shows a flow chart of an ExceptionPending routine executed by an Exception Detection function forming part of the M68k emulator kernel of Figure 3;

Figure 24 shows a flow chart and a data flow diagram of an AutoVector routine executed by the Exception Detection function forming part of the M68k emulator kernel of Figure 3;

Figure 25 shows a flow chart and a data flow diagram of an Exception Handler function forming part of the M68k emulator kernel of Figure 3;

Figure 26 shows a flow chart of a PackSR routine executed by the M68k emulator kernel of Figure 3;

Figure 27 shows a flow chart of a UnpackSR routine executed by the M68k emulator kernel of Figure 3;

Figure 28 shows a flow chart and data flow diagram of an Emu68kInit routine executed by a control component of the emulation API of Figure 4;

Figure 29 shows a flow chart and data flow diagram of an Emu68kRun routine executed by the control component of the emulation API of Figure 4;

Figure 30 shows a flow chart and data flow diagram of an Emu68kStep routine executed by the control component of the emulation API of Figure 4;

Figure 31 shows a flow chart and data flow diagram of an Emu68kWriteSRecord routine executed by a memory access component of the emulation API of Figure 4;

Figure 32 shows a flow chart and data flow diagram of an Emu68kProcessSRecord routine executed by the memory access component of the emulation API of Figure 4;

Figure 33 shows a sample flow chart and sample data flow diagram of an "Emu68kRequest..." routine executed by a requests component of the emulation API of Figure 4; and

Figure 34 shows a flow chart and data flow diagram of an Emu68kSystemsExecptions routine executed by the requests component of the emulation API of Figure 4.

Detailed Description Of The Preferred Embodiments

Referring now to Figure 1, an emulation system to allow code written for a specific computing environment to be run on a different microprocessor platform is shown and is generally indicated to by reference numeral 50. In the preferred embodiment, the emulation system 50 allows Motorola 68000 ("M68k") code to be emulated and run on an Intel 80x86 microprocessor platform. The emulation system 50 forms part of a PC-based PBX (private branch exchange) incorporating pre-existing call control software written in SX2000 code for a non-PC-based environment as described in co-pending Canadian application serial No. _____ filed on May 1, 1998 for an invention entitled "Method And Apparatus For Migrating Embedded PBX System To Personal Computer". In this manner, the pre-existing call control software can be maintained even though the processing platform is different. Those of skill in the art will however appreciate that the emulation system can be used in other environments and can be configured to provide for the emulation and running of M68k code or other code on different microprocessor platforms.

The emulation system 50 includes an emulator generator 52 accessing an external data-table 54, a M68k emulator 56 communicating with the emulator generator 52 and an emulation applications program interface (API) 58 communicating with the M68k emulator 56. The emulator generator 52 includes a large table (hereinafter referred to as a static string structure) containing emulation code segments written in Intel assembly which can be assembled to create emulation routines. The emulation routines when executed by the M68k emulator 56 allow for M68k instructions to be carried out.

The emulation code segments are designed to replace each M68k instruction with a set of Intel instructions that perform an identical task and support Motorola-like registers. The M68k instruction set consists of fifty-five base

operations, some of which include additional variations or subsets. With few exceptions, operations performed on bytes, words and long words, can use any of the 14 Motorola addressing modes and work with any of the 14 Motorola registers. Combining instruction types, data types, addressing modes and source/destination registers produces several thousand useful instructions.

The instruction set of the M68k microprocessor is based on a 16-bit instruction format. Thus, each instruction is represented by a specific 16-bit pattern, comprised of an instruction mask and permutation masks. The permutation masks are required to identify addressing modes, affected registers and size of the operations. Of the sixty-five thousand possible permutations available to a 16-bit format, about forty-seven thousand are used. The remaining combinations were not used by Motorola as of the day the present application was filed. The static string structure contains emulation code segment entries corresponding to the forty-seven thousand M68k instructions, each entry being between 5 and 50 lines long.

The M68k emulator 56 executes the emulation routines created by the emulator generator 52 to carry out the M68k instructions while the emulation API 58 provides an interface as similar as possible to that of the M68k microprocessor platform. In this manner, the emulation system 50 appears as a M68k microprocessor to the pre-existing software being executed.

The external data-table 54 contains a list of frames or characteristic data packets representing all of the M68k instructions to be emulated. The stored characteristic data packets describe the syntax, base operation, operand size, instruction format, source addressing modes available, destination addressing modes available, condition codes modified by instruction, privilege level, modification of source and/or destination, change in flow, and possible modifications to status register, of each instruction. Separate frame entries are created in the external data-table 54 to distinguish between instructions as a function of the operation size. That is, frames are created for byte operations, for word operations and for long word operations. Multiple frames entries are created, if required, to differentiate amongst variations or subsets of a given instruction. For example, frames entries are created

for the ADD instruction and the ADDI (add immediate) instruction which is a subset of the ADD instruction. Just over 300 frame entries provide all of the characteristic information required to generate the full set of M68k instructions to be emulated.

Figure 2 better illustrates the emulator generator 52 and as can be seen, the emulator generator includes an input component 60, an emulation routine generation component 62 and an output component 64. The input component 60 is responsible for reading information on each M68k instruction to be carried out from the external data-table 54 and for initializing the static string structure as well as an emulation structure as will be described. The emulation routine generation component 62 is responsible for determining the characteristics of each M68k instruction to be carried out and based on the determined characteristics, selecting appropriate emulation code segments stored in the static string structure. The output component 64 is responsible for piecing the emulation code segments together to create emulation routines 66 so that each M68k instruction will be emulated when the corresponding emulation routine is run by the M68k emulator kernel 56. The output component 64 is also responsible for generating jump-tables 68 containing linking addresses to the emulation routines to allow the M68k emulator 56 to access the emulation routines. These emulation routines 66 and jump-tables 68 are generated once and included as relocatable objects during link time with the M68k emulator kernel 56 code.

Referring now to Figure 3, the M68k emulator kernel 56 is better illustrated. As can be seen, the M68k emulator kernel includes an emulator loop 70 executing a main program and accessing memory 72, an exception detection function 74 and an exception handler function 76. Two supporting routines are executed by the M68k emulator kernel 56, namely a PackSr routine and an UnpackSr routine as will be described.

The main program loop is basically a continuous loop which reads M68k instructions from the memory 72, uses the M68k instructions to index into the jump-tables which contain the location of the corresponding emulation routine associated with the read M68k instructions, executes the emulation routines and then

cycles back to repeat the process. The main program is exited only when an exception is present and is detected by the emulator loop 70. The exceptions may be software (S/W) exceptions, hardware (H/W) interrupts, trace exceptions or system exceptions as determined by the exception detection function 74 and are treated by the M68k emulator kernel 56 as requests. When software or trace exceptions or hardware interrupts are detected, the main program is exited and control is passed to the exception handler function 76 to allow the exceptions to be handled. Once the exception handler function 76 handles the exceptions, control is passed back to the main program of the emulator loop 70. When a system exception is detected, the M68k emulator kernel 56 is exited. A new call to the M68k emulator kernel 56 must be made in order to return to the main program loop.

The emulator API 58 is best illustrated in Figure 4, and as can be seen, the emulator API includes a control component 80, a registers component 82, a requests component 84 and a memory access component 86. Control, registers and requests components 80 to 84 respectively communicate with the M68k emulation kernel 56 while memory access component 86 communicates with the memory 72.

The control component 80 is responsible for initializing and cleaning-up the M68k emulator kernel 56 and for conditioning the M68k emulator kernel 56 to operate in either a step mode or a run mode as will be described. The memory access component 86 provides an interface to the memory 72 allowing M68k instructions to be loaded into the memory 72 once the M68k emulator kernel 56 has been initialized. The M68k instructions are written to the memory 72 in an S-Record format (see Appendix A) using a Write-S-Record function.

The registers component 82 makes the registers that are available to a native M68k microprocessor platform available to the M68k emulator kernel 56. The registers made available to the M68k emulator kernel 56 by the registers component 82 include control registers such as the status register (SR), the program counter (PC), the user stack pointer (USP) and the supervisor stack pointer (SSP), information registers such as the data registers (D0, D1, D2 .. D7) and the address registers (A0,

A1, A2 .. A7). The M68k emulator kernel 56 also supports a vector base register (VBR) used in the native M68020 instruction set.

The requests component 84 allows requests (i.e. software exceptions, hardware interrupts, trace exceptions and system exceptions) to be set and cleared and their status tested.

In the present embodiment, the main function of the emulation system 50 is to take existing SX2000 code (which is designed to take advantage of the M68k microprocessor architecture and is thus very tightly coupled to this architecture) and run it on an Intel 80x86 platform. Due to the different processor structures (register structure, instruction set, etc.), each M68k instruction must be translated into and emulated by a series of Intel 80x86 instructions.

The emulator generator 52 is run to generate the emulating routines 66 and the jump-tables 68. The emulating routines 66 and jump-tables 68 are assembled (currently using Microsoft Assembler 5.1) to generate object files to be linked with the M68k emulator kernel 56 and emulation API 58.

During start-up, the SX2000 code is loaded from file and stored into the memory 72 using the S-Record format (see Appendix A). This format specifies where within the memory each section of the code is to be located. The S-Record is read directly as a file. In order to run the SX2000 code, the M68k emulator kernel 56 takes each M68k instruction read from memory 72 and uses it to index into the jump-tables 68 to access the emulation routine. The emulation routine is then executed to carry out the M68k instruction. Once the M68k instruction has been processed, the M68k emulator kernel 56 tests for exceptions and if none are found automatically executes the subsequent M68k instruction. This process continues indefinitely unless exception handling requires exceptions to be processed. The emulation API 58 protects the M68k emulator kernel 56 and provides an interface resembling the M68k microprocessor environment. Further specifics of the emulation system 50 and its operation will now be described.

Emulator Generator

The emulator generator 52 works on the assumption that various code segments, each performing a specific function, can be integrated to create smoothly operating routines. To achieve this, recurring functionality was assigned to specific registers in the Intel space. The table below lists the "dedicated" registers and their assigned function.

Dedicated **Intel** Registers (size long)
EAX = destination data
EBX = pointer to destination data
ECX = source data
EDX = pointer to source data
EBP = pointer to base memory

Input Component

Figures 5 and 6 better illustrate the input component 60 and as can be seen, the input component includes a number of functions, namely an initialize emulation function 100, an initialize static strings function 102 and a read operation format function 104. The initialize emulation function 100 pre-sets the emulation structure into a known state by setting pointers within the emulation structure to Null. The initialize static strings function 102 initializes strings (i.e. emulation code segments) in the static string structure. The static string structure contains a significant number of strings containing just about all the required code segments to create all of the M68k instructions. The static string structure is filled during initialization. The fields that are available in the static string structure are described individually in the following sections.

paszOpMnemonics

paszOpMnemonics [opMAX];

array is indexed by [operation enumeration: ABCD.B =0, ADD.B =1, etc.];

this field of the structure is an array of strings. Each string contains the mnemonic of an operation. The mnemonic string matches the value used in the frame format structure operation field to describe the operation. The index into this string is the operation number (which is a value from an enumerated list).

paszSrcAddrStr

paszSrcAddrStr [OP_SIZE_MAX][0100];

array is indexed by [operation size: byte, word, long] [operation address mode: (An), (d_{16},An), (xxx.W), etc.];

The addressing mode is in octal format and adheres to the following standard:

for the range 000 to 067 (octal values) the value XY indicate the mode

X is used to identify the addressing mode in use	**Y** is used to identify the register used
0 - Dn	0 - 0 (such as D0 or A0)
1 - An	1 - 1 (such as D1 or A1)
2 - (An)	2 ..
3 - (An)+	3 ..
4 - -(An)	4 ..
5 - (d16,An)	5 ..
6 - (d8,An,Xn)	6 ..
	7 - 7 (such as D7 or A7)

The values in the 07z range indicate the following addressing modes:

70 - (xxx.W)

71 - (xxx.L)

72 - (d_{16},PC)

73 - (d_8,PC,Xn)

74 - <data>

75 - (BYTE)data16

76 - <data16>

77 - <data32>

the array contains code segments used to support retrieving source data when not directly from registers - i.e. utilizing memory addressing modes or special addressing modes. The code moves the data address into the Intel register reserved for source data address and correctly updates the emulated registers (i.e. for the modes (An)+, -(An), etc.)

paszReadSrcString

paszReadSrcString [OP_SIZE_MAX][0100];

array is indexed by [operation size: byte, word, long] [operation address mode: (An), (d_{16},An), (xxx.W), etc.];

(for details on operation addressing mode refer to pasrSrcAddrStr);

the array contains code segments used to support retrieving source data. The code moves the data to the Intel register reserved for source data. This will vary depending on the addressing mode (for direct registers the data is taken from register, otherwise it is read from the address previously set-up).

paszSrcImpliedAddrString

paszSrcImpliedAddrString [0x100];

array is indexed by [implied number: 0-255];

the array contains code segments to support source implied address mode.

paszWriteSrcString

paszWriteSrcString [OP_SIZE_MAX][0100];

array is indexed by [operation size: byte, word, long] [operation address mode: (An), (d_{16},An), (xxx.W), etc.];

(for details on operation addressing mode refer to section paszSrcAddrStr);

the array contains code segments used to support updating source data following the execution of the instruction. The code moves the data from the Intel register reserved for source data to the variable representing the Motorola register.

paszDstAddrStr

paszDstAddrStr [OP_SIZE_MAX][0100];

same as paszSrcAddrStr description but for destination data.

paszReadDstString

paszReadDstString [OP_SIZE_MAX][0100];

same as paszReadSrcString description but for destination data.

paszWriteDstString

paszWriteDstString [OP_SIZE_MAX][0100];

same as paszWriteSrcString description but for destination data.

paszOpCodeString

paszOpCodeString [opMAX][OP_SIZE_MAX];

array is indexed by [operation enumeration: ABCD.B =0, ADD.B =1, etc.] [operation size: byte, word, long];

A limited number of operations are recognized by the Emulator Generator as being the core operations and they all have supporting operation-specific code. All other functions can be generated using these base operations. For example ADD.B is a base operation, ADDA.W is a base operation, but ADDI.B is not and it can be generated using ADD.B. This array contains the code segments to support this operation-specific functionality.

paszCcrString

paszCcrString [ccrMAX];

array is indexed by [ccr-operation enumeration: clear_C, clear_V, clear_Z, ... , set_C, set_V, ... , test_C,..etc.];

contains the code segments to support clearing, setting and testing of the Condition Code Registers.

ppszSrPackStruct

code segment to pack the Status Register structure - calls the PackSR routine of the Emulator (see reference document).

ppszSrUnpackStruct

code segment to unpack the Status Register structure - calls the UnpackSR routine of the Emulator (see reference document).

ppszCheckPcString

code segment to test the Program Counter for an odd value if so cause an address error exception;

following a change in PC by the instruction, the PC must be tested to ensure that it does not rest on an odd boundary as this would cause an address error;

Note: the PC is incremented within the M68k Emulator Kernel (see Emulator reference) by incrementing its value to the next word boundary. An operation may require that the PC be modified further due to the presence of data following the instruction. For example the data used in an ADD immediate follows the instruction, or the address information stored immediately following a branch instruction. In such cases the PC is modified accordingly within the code of the emulating routine. The information is contained within the paszOpCodeString field.

ppszCheckPrivilegeString

code segment to test the operating mode (supervisor or user) to determine if the operation can be executed. If operating mode is not supervisor, set a privilege violation exception;

Some operations are allowed to be executed only while in supervisor mode due to their ability to affect a whole system (such as RTE, RESET, ORI to SR, MOVES, etc.).

ppszVarDefinitionString

code segment which contains a list of all the external variables used to emulate Motorola 68000 registers and other microprocessor functions.

As mentioned above, the initialize static strings function 102 initializes code segments stored in the static string structure by calling a number of routines (see Figures 7 and 8) to perform the initialization. In particular, the initialize static strings function 102 initializes the code segments used for: reading source data from registers; reading the data using various forms of addressing; updating the destination data using various forms of addressing, etc. As part of the static string structure initialization, conditional code register (CCR) strings are also initialized with the code necessary to support clearing, setting and testing the C-carry, V-overflow, Z-zero, N-negative and X-extend bits of the conditional code registers. Conditional code information is contained in bits within the status register as in the native M68k microprocessor as well as in separate conditional code variables, one variable per bit, for ease of processing.

The initialize static strings function 102 also initializes the strings required for the packing and unpacking of the status register (SR) structure entries, the strings required for testing instruction privilege (user or supervisor mode) and the code segments for testing possible address violations following modifications to the program counter (PC). In addition, the initialize static strings function 102 initializes the strings required for operation specific code.

To initialize the static string structure as described, the initialize static strings function includes a Load Instruction Mnemonics routine 102a; an Initialize Source Address routine 102b; an Initialize Source Read Register routine 102c; an Initialize Source Write Register routine 102d; an Initialize Destination Address routine 102e; an Initialize Destination Read Register routine 102f; an Initialize Destination Write Register routine 102g; an Initialize Operation Specific Code routine 102h; an Initialize Conditional Code Registers routine 102i; and an Initialize SR, PC, Privilege, Variable Strings routine 102j.

The Load Instruction Mnemonics routine 102a loads the mnemonics of the supported operations. The Initialize Source Address routine 102b initializes code segments to support address modes. The Initialize Source Read Register routine 120c initializes code segments to support reading from source registers. The Initialize Source Write Register routine 102d initializes code segments to support writing data to source following instruction. The Initialize Destination Address routine 102e initializes code segments to support destination address modes. The Initialize Destination Read Register routine 102f initializes code segments to support reading data from destination registers. The Initialize Destination Write Register routine 102g initializes code segments to support writing data to destination following instruction. The Initialize Operation Specific Code routine 102h initializes code segments supporting the specifics of the various operations. The Initialize Conditional Code Registers routine 102i initializes code segments supporting setting, clearing and testing CCRs. The Initialize SR, PC, Privilege, Variable strings routine 102j initializes code segments supporting the status register SR, tests for program counter PC, tests for Privilege and providing register Variable definitions.

The read operation format function 104 is called once the emulation structure and static string structure have been initialized and includes a read frame format routine 106 and a parse frame format routine 108. The read operation format function 104 retrieves frames from the external data-table 54 that are required for each emulation routine to be created. The steps performed by the read operation format function are better shown in Figures 9 and 10. As can be seen, when the read operation format function 104 is called, the read operation frame format routine 106 retrieves the frames from the external data-table 54 that are necessary for the M68k instruction to be carried out. As each frame is read from the external data-table 54, the read frame format routine 106 saves the information of the read frame as an entry in a format frame structure. The parse frame format routine 108 then acts on the saved information to remove spaces, tabs and comments from the entry. These steps are performed until all of the necessary frames have been read from the external data-table 54.

The format frame structure contains the formatting information for all operations to be emulated. The format frame structure is filled with information from the external data-table 54. The fields that are available in the format frame structure are described individually in the following sections.

SYNTAX

The syntax field provides an opportunity to define what operation is defined by the frame. The field is only used to differentiate the operations and for display purposes to ease trouble-shooting. The field entry consists of the operation and operation's use:

ADD.B <ea>,Dn or

ADD.B Dn, <ea> (two different operations).

OPERATION

A limited number of operations are recognized by the emulator generator 52 as being the core operations and they all have supporting operation-specific code. All other functions can be generated using these base operations. For example ADD.B is a base operation, ADDA.W is a base operation, but ADDI.B is not and it can be generated using ADD.B.

The field contains the operation whose code will be used to support the functionality.

A list of all base-operations is provided during initialization by the Load Instruction Mnemonics routine 102a.

SIZE

size of the operation .... byte, word, long, unsized (for operations which are not affected by size like bkpt, trap, rts or reset) or word:long (for operations in which the source is word but the destination is long such as adda.w, cmpa.w, divs.w, muls.w, etc.).

FORMAT

the instruction format, a binary representation of the 16-bit instruction. It includes the fixed component of the operation as well as the source, destination or

effective addressing modes. The fixed component is expressed using 0s and 1s, the source and destination modes are described using the following categories:

ddd - source Dn

aaa - source An

nnn - source number 3 bit (1-7 and 0 -> 8)

nnnnnnnn - source number 8 bit

NNNNNNNN - source number 8 bit 1..255 (!= 0)

mmmrrr - source Mode:Register

DDD - destination Dn

AAA - destination An

MMMRRR - destination Mode:Register

For example for the DIVS.W instruction:

format is: 1000 DDD0 11mm mrrr

destination: DDD

source: mmmrrr

SRC_ADDR_MODE

describes the valid addressing modes for the source component using a 16-bit representation (unrelated to the format). The following addressing modes are supported:

0	Dn	- Data register direct
1	An	- Address register direct
2	(An)	- Memory address - address register indirect
3	(An)+	- Memory address - address register indirect with postincrement
4	-(An)	- Memory address - address register indirect with predecrement
5	(d_{16},An)	- Memory address - address register indirect with displacement

6 (d_8,An,Xn) - Memory address - address register indirect with index

7 enable special addressing modes

8 (xxx.W) - Special - absolute short address

9 (xxx.L) - Special - absolute long address

10 (d_{16}, PC) - Special - program counter with displacement

11 (d_8,PC,Xn) - Special - program counter with index

12 <data> - Special - immediate data

13 (BYTE) data16 - Special - immediate data (lower byte of 16 bit value)

14 <data16> - Special - immediate data (16 bits)

15 <data32> - Special - immediate data (32 bits)

The first number on the above table is the bit position representing the addressing mode (starting from the most significant bit!).

For example for the DIVS.W instruction:

source addressing mode : 1011 1111 1111 1000

destination addr. mode : 1000 0000 0000 0000

meaning the source supports Dn, all Memory address modes, and (xxx.W), (xxx.L), (d_{16},PC), (d_8,PC,Xn) and <data> of the special modes. The destination only supports Dn mode.

DST_ADDR_MODE

refer to the description in SRC_ADDR_MODE (§0).

CCR

this 5 character field describes the effects of the conditional code registers and the required support. The characters represent: X (extend), N(negative), Z(zero), V(overflow) and C (carry) in that order from left to right. The supported modes are:

* for testing,

0 for clear,

1 to set,

and - and U to leave the register untouched

For example for the DIVS.W instruction:

-***0 => indicates NZV to be tested/updated, C to always reset and X remains untouched.

SR_PACK

this field contains either a "yes" or "no" to indicate whether packing of the status register is required or not.

Packing or Unpacking of the status register is used when the operation leads to a change in the information contained in the status register (mode, trace, interrupt mask, conditional codes) or when the operation requires the information.

SR_UNPACK

this field contains either a "yes" or "no" to indicate whether unpacking of the status register is required or not.

Packing or Unpacking of the status register is used when the operation leads to a change in the information contained in the status register (mode, trace, interrupt mask, conditional codes) or when the operation requires the information.

PRIVILEGE

this field contains either a "yes" or "no" to indicate whether the operation is restricted and as such must be executed exclusively under supervisor mode. (for example RTE, RESET, ORI to SR, etc.).

GET_SRC

this field contains either a "yes" or "no" to indicate whether the operation requires to collect source data. Most operations require source data, but some do not (such as TRAP, SF, RTS, etc.). Some require destination without source (such as SF).

GET_DST

this field contains either a "yes", "no" or "implied" to indicate whether the operation requires to collect destination data. Most operations require destination data, but some do not (such as TRAP, RTS, etc.). Some require destination without source (such as SF).

SRC_ACTIVE

field contains either a "yes" or "no" to indicate whether the source data is active - that is if it requires updating at the completion of the instruction (i.e. has been modified). For example the operation EXG.W which exchanges the contents of the two registers requires modification of the source. (For more information on the EXG operation consult the Motorola reference).

DST_ACTIVE

field contains either a "yes", "no" or "implied" to indicate whether the destination data is active - that is if it requires updating at the completion of the instruction (i.e. has been modified). Many operations fall in this category ... for example the operation MOVE.B requires modification of the destination.

PC_CHECK

field contains either a "yes" or "no" to indicate whether the operation has modified the program counter (PC) and requires to test for possible address errors (odd PC).

FIELD_MAX

this field is simply the last enumerator for the various fields. It is not modified.

Emulation Routine Generation Component

Turning now to Figures 11 to 13, the emulation routine generation component 62 is better illustrated. As can be seen, the emulation routine generation component includes a determine operation characteristic function 120, a determine source mask function 122, a determine destination mask function 124 and a fill

emulation string table function 126. The determine operation characteristic function 120 accesses the format frame structure and the static string structure to determine the crucial characteristics for each operation to be performed to carry out the M68k instruction and stores these operation characteristics in an operation characteristics structure.

The fields that are available in the operation characteristics structure are described individually in the following sections.

nOperation

operation number as given by the operation's position within an enumerated list;

used as an index to retrieve information from the static string structure paszOpCodeString field.

nSrcSize

source operation-size as a number given by the size's position within an enumerated list (i.e. byte = 0, word = 1, long = 2, etc.);

the size is used as an index to retrieve information from various fields of the static string structure.

nDstSize

destination operation-size as a number given by the size's position within an enumerated list (i.e. byte = 0, word = 1, long = 2, etc.);

the size is used as an index to retrieve information from various fields of the static strings structure.

wInstrMask

operation mask;

calculated from the format frame structure format field;

the operation mask is the fixed component of the instruction format. The fixed component of the binary representation of the 16-bit instruction (i.e. no source or destination permutations information).

For example for the DIVS.W instruction:

format is: 1000 DDD0 11mm mrrr

operation mask: 1000 0000 1100 0000

wSrcAddrMode

source address mode;

determined from the format frame format field;

the source and destination modes are described below along with the corresponding enumerated "key" value (taken from a local list) which is used internally:

Mode	Key	Address mode
ddd	- Data_Reg	- source Dn
aaa	- Addr_Reg	- source An
nnn	- Num_3Bit	- source number 3 bit (1-7 and 0 ->8)
nnnnnnnn	- Num_8Bit	- source number 8 bit
NNNNNNNN	- Num_8Bit_NonZero	- source number 8 bit 1..255 (!= 0)
mmmrrr	- Mode_Reg	- source Mode:Register
DDD	- Data_Reg	- destination Dn
AAA	- Addr_Reg	- destination An
MMMRRR	- Mode_Reg	- destination Mode:Register
RRRMMM	- Reg_Mode	- destination Register:Mode

For example for the DIVS.W operation:

format is: 1000 DDD0 11mm mrrr

wSrcAddrMode: Mode_Reg (actual value given by enumeration list)

wSrcAddrOffset

offset of the source address mode;

the distance (in bits) that the least significant bit of the value representing the address mode is located from the least significant bit of the operation format;

For example for the DIVS.W operation:

format is: 1000 DDD0 11mm mrrr

wSrcAddrOffset: 0

wDstAddrOffset

same as wSrcAddrMode but for destination address mode

For example for the DIVS.W operation:

format is: 1000 DDD0 11mm mrrr

wDstAddrMode: Data_Reg

wDstAddrMode

same as wSrcAddOffset but for destination address mode

For example for the DIVS.W operation:

format is: 1000 DDD0 11mm mrrr

wDstAddrOffset: 9

abSrcPermission

abSrcPermission [8];

array reflecting the various addressing modes available for source addressing. A mode is available if the respective byte is set (1), not available if reset (0);

bytes 0 to 6 represent direct register and memory addressing modes, bits within byte 7 represent availability of special addressing modes;

the supported addressing modes are:

byte	addressing mode	
0	Dn	- Data register direct
1	An	- Address register direct
2	(An)	- Memory address - address register indirect

3 (An)+ - Memory address - addr. regi. indirect with postincrement

4 -(An) - Memory address - addr. reg. indirect with predecrement

5 (d_{16},An) - Memory address - addr. reg. indirect with displacement

6 (d_8,An,Xn) - Memory address - addr. reg. indirect with index

bit of [7]	addressing mode	
1	(xxx.L)	- Special - absolute long address
2	(d_{16}, PC)	- Special - program counter with displacement
3	(d_8,PC,Xn)	- Special - program counter with index
4	<data>	- Special - immediate data
5	(BYTE) data16	- Special - immediate data (lower byte of 16 bit value)
6	<data16>	- Special - immediate data (16 bits)
7	<data32>	- Special - immediate data (32 bits)

abDstPermission

abDstPermission [8]

same as abSrcPermission but for destination addressing modes;

The source mask function 122 determines the source mask and source mode register specific to the M68k instruction while the destination mask function 124 determines the destination mask and destination mode register specific to the instruction. Once the M68k instruction has been determined (block 128), the fill emulation string table function 126 populates an emulation string structure with the proper code segments taken from the static string structure. If the M68k instruction

format specifies that the functionality is not required, then the entry in the emulation string structure is left void. The above steps are performed for each destination permutation, source permutation and operation (blocks 130 to 134 respectively).

The fields that are available in the emulation string structure are described individually in the following sections.

pszOperationString

instruction's copy of information from paszOpMneumonics.

pszCheckPrivilege

instruction's copy of information from ppszCheckPrivilegeString.

pszGetSrcAddr

instruction's copy of information from paszSrcAddrStr.

pszGetSrcData

instruction's copy of information from paszReadSrcString.

pszGetDstAddr

instruction's copy of information from paszDstAddrStr.

pszGetDstData

instruction's copy of information from paszReadDstString.

pszOpCode

instruction's copy of information from paszOpCodeString.

pszSaveCCR

code segment to save the Intel's flags (comparable to conditional code registers).

pszRestoreCCR

code segment to restore the Intel's flags (comparable to conditional code registers).

pszCcrX

instruction's copy of information frompaszCcrString.

pszSrPack

instruction's copy of information from ppszSrPackStruct.

pszSrUnpack

instruction's copy of information from ppszSrUnpackStruct.

pszPutSrcAddr

instruction's copy of information from paszWriteSrcString.

pszPutDstAddr

instruction's copy of information from paszWriteDstString.

pszCheckPc

instruction's copy of information from ppszCheckPcString,

When the determine operation characteristic function accesses the format frame structure and the static string structure to determine the crucial characteristics for each operation to be performed, the operation characteristic function 120 determines the source addressing mode and the destination addressing mode (register direct, memory indirect, special modes, etc.) from the operation mask. The position of the addressing modes is also located within the 16-bit instruction format (i.e. the bits used to represent the permutation available to the instruction to indicate which addressing modes are being used). Knowing the location of the permutations determines how the modes are supported within the instruction format.

Each operation has characteristics which determine what code segments are utilized to populate the emulation string structure. Some characteristics are also used to determine the source or destination masks. The operation characteristics function 120 therefore calls a number of routines to determine the code segments to be utilized as shown in Figures 14 and 15. Initially, the operation characteristics function 120 calls a Determine Operation routine 120a to determine

what operation-specific code must be included. A Determine Operation Size routine 120b is then called to determine the size of the operation for both the source and the destination. In most cases this will be identical, but in a few instances, such as for operations which deal with addresses ADDA, the source might be different than the destination (word for source, long for destination). A Determine Operation Mask routine 120c is then called to determine the operations mask. It is important to know the operations mask as it describes the constant section of the 16-bit format. Get Source Addressing Mode and Get Destination Addressing Mode routines 120d and 120e respectively are then called to determine the source and destination address modes and offsets from the operations format. For example, for the operation ABCD with format 1100 DDD1 0000 0ddd, the following would be calculated:

operations mask:	1100 0001 0000 0000
source mode:	DDD offset 0 (to indicate source from Dn registers)
destination mode:	DDD offset 9 (to indicate destination from Dn registers)

Get Source Permission and Get Destination Permission routines 120f and 120g respectively are then called to determine which of the addressing modes are supported by the operation (Dn, An, (An), (An)+, etc.).

As mentioned previously, the fill emulation string table function 126 is responsible for populating the emulation string structure with correct code segments for each specified instruction. The fill emulation string table function 126 therefore calls a number of routines during population of the emulation string structure as shown in Figures 16 to 18. Firstly, when the fill emulation string table function 126 is invoked, it calls a Test For Collisions routine 126a which performs a test to ensure that no other operation has generated this particular instruction. A severe program error would result if this event were to occur. A Get Source String routine 126b is then called which determines the code segments necessary to support reading the

source data taking the source address mode into consideration (i.e. retrieve data using various forms of addressing). This process is then repeated for the destination data by calling a Get Destination String routine 126c.

Some operations are only allowed to function under the supervisor mode of operation of the microprocessor. Thus, some operations require that the privilege of the operation be tested prior to performing the operation. The fill emulation string table function 126 therefore calls a Privilege String routine 120d to support the above. A Program Counter String routine 126e is then called to test for address errors in any operation that modifies the program counter (PC). A Conditional Code Register String routine 126f is then called to support conditional code registers CCRs. An Operation Specific Code routine 126g is then called to include the code specific to the operation and then an Update Source/Destination Code routine 126h is called to support any required updating of the source or destination data as a result of the operation.

Output Component

Figures 19 to 21 better illustrate the output component 64 and as can be seen, it includes an emulation code function 150 and a jump-table information function 152. The emulation code function 150 pieces the code segments in the emulation string structure together to create emulation routines 66 and writes the emulation code segments to the emulation code structure.

The fields that are available in the emulation code structure are described individually in the following sections.

nSrcMask

source mask - 16-bit value.

a permutation of the source addressing component within the operation's format which specifies the instruction's characteristics. The value exists only if the addressing mode specified by the operation characteristics is supported.

For example for the DIVS.W operation:

format is: 1000 DDD0 11mm mrrr

source addr. mode mmmrrr (this is a 6 bit field - covering 64 permutations)

assume current source permutation is: 010011 (binary)

if this addressing mode (represented by the top 3 bits - 010 in this case => mode (An)) is supported by the operation (as determined by the source permission array abSrcPermission) then the source mask would be

source mask: 0000 0000 0001 0011

nSrcModeReg

source mode and register;

the value (00 to 77 - octal) is a combination of the addressing mode (first octal digit) and the register implemented by that mode (second digit). The values of 070 to 077 represent the special addressing modes (thus no value for register).

nDstMask

as in nSrcMask but for destination field.

nDstModeReg

as in nSrcModeReg but for destination field

The jump-table information function 152 creates the jump-tables 68 to allow the emulation routines to be accessed by providing linking addresses to the emulation routine using the M68k instruction number as an index.

Out of the possible 65,000 permutations available with a 16-bit frame, about 47,000 permutations identify valid M68k instructions. Considering that the emulation routines cover between 5 and 60 lines of code each, generating 47,000 emulation routines requires significant memory space. To limit assembly and link times, the emulation routines output by emulator generator 52 and stored in the emulation code structure are divided and stored into 16 assembly files. The value of the first nibble of the instruction is used to determine in which file the code of each emulation routine will be placed. File names are of the format "i?xxx.asm", where ? refers to the value of the first nibble. For example, all routines emulating instructions

starting with a high nibble value of 4 are placed in the file "i4xxx.asm" (instructions for xxx). Although all of the required information is placed into the emulation structure by the emulator generator 52, the emulation routines are not entities until the code segments are organized as they are written to file.

Generating the emulation routines by utilizing only the required components for each instruction is a task performed by a Write Single Instruction To File routine 154 called by the emulation code function 150 . This routine calls a Print Data If Not Null routine 156 to print the information to file. The Write Single Instruction To File routine 154 organizes emulation code segments for given instructions and writes the information to file. The Print Data If Not Null routine 156 writes data to the file if the pointer to the data is valid.

Forty-seven thousand instructions constitutes a considerable number of emulation routines to link. To provide quick access to the emulation routines, the jump-tables 68 are used. As each emulation routine is created, an entry in the jump-tables 68 is generated to provide the link address of an emulation routine by using the M68k instruction number as an index into the jump-table 68. That is, the jump-table is indexed with a 16-bit instruction and contains the starting address of the emulation routine. In practical terms, the jump-table 68 holds a label of the emulation routine. The label is resolved into the starting address of the emulation routine during the linking stage. To limit link time and to reduce the size of the jump-table 68, the jump-table is also divided into 16 assembly files. The value of the first nibble of the instruction is used to determine in which assembly file the jump information is to be included. File names use the format "jmptable?.asm", where ? refers to the value of the first nibble.

M68k Emulator Kernel

Emulator Loop

As mentioned earlier, the M68k emulator kernel 56 executes a continuous emulator loop 70 best seen in Figure 22 allowing emulation routines to be fetched and executed to carry out M68k instructions. Access to the main program of the emulator loop 70 is made via a call to Emu68000. When the call is made, access

to memory 72 is prepared by setting the base pointer (block 180). The stack pointer that is to be used given the current operating mode (supervisor or user) is then determined (block 182). The UnpackSR routine is then invoked to update the Intel flags using the status register SR (block 184). Following this, the program counter is loaded and a M68k instruction is read from the memory 72. The program counter is then updated and the jump-table 68 is used to find the address of the emulation routine in the emulation code structure to be run to carry out the read M68k instruction. A jump is then made to the emulation code allowing the emulation routine to be executed (blocks 186 and 188). Following this, the main program loop checks to determine if any exceptions have been received (blocks 190 to 194) or if any exceptions are pending (block 196). If no exceptions have been received or are pending, the main program returns to block 186. If an exception has been received or is pending, the emulator loop 70 is exited and control is passed to the exception detection function 74.

Exception Detection

The exception detection function 74 includes two components, namely an ExceptionPending routine and an AutoVector routine. When control is passed to the exception detection function 74, the ExceptionPending routine is called (see Figure 23). The ExceptionPending routine firstly updates the flags register (IntelSR) (block 200). Thereafter, the PackSR and UnpackSr routines are called (blocks 202 and 204). Following execution of the UnpackSR routine, the ExceptionPending routine determines whether the detected exception is a software exception (block 206), a hardware interrupt (block 208), a trace exception (block 210) or a system exception (block 212). This is necessary since each exception is treated differently.

If a software exception or a trace exception is detected, the ExceptionPending routine passes control to the Exception Handler function 76 so that the exceptions can be handled. If a hardware interrupt is detected, the ExceptionPending routine calls the AutoVector routine.

Referring now to Figure 24, when the AutoVector routine is called, the AutoVector routine checks to determine which hardware interrupt is set (blocks 220 to 232). Once the set hardware interrupt is known, the AutoVector routine sets the appropriate internal variables, namely the currentException, the interruptMask and the interruptMask bit SR_I (block 234). If the hardware interrupt cannot be determined, a spurious interrupt exception is set 236. Following this, the AutoVector routine calls the Exception Handler function 76.

Exception Handler

When the Exception Handler function 76 is called, before the exception handling process is emulated, the exception handler function 76 saves the registers (block 240) and then checks to see if the exception is bound to routines to be executed in native Intel mode (block 242) (see Figure 25). This can be used for documentation of error, trigger for the execution of various processes etc. If a bound routine to the exception vector function table is present, it is executed (block 244). If a bound routine is not present or after the bound routine has been executed, the Exception Handler function 76 restores the registers (block 246). A check is then made to determine if the supervisor mode has been set (block 248). If not, the supervisor mode is set, the user stack pointer USP is saved and the supervisor stack pointer SSP is used (block 250). Thereafter, the Exception Handler function 76 prepares the M68k emulation kernel 56 to process the exception. In particular, the Exception Handler function 76 saves the status register, the program counter (prior to modification to support execution) and the stack format on the stack for retrieval at completion of the exception. The exception number is then used to index into the vector base table that contains the addresses of the exception handling routines. The program counter is updated, the exception variables are cleared and the M68k emulation kernel 56 returns to the emulator loop 70. Blocks 252 to 266 reflect the above steps.

PackSR and UnpackSR

The PackSR and UnpackSR routines are used extensively by the M68k emulation kernel 56 and emulation routines 66 and are responsible for transferring status flag information from the Intel platform to the Motorola platform (PackSR) and vice versa (UnpackSR). The routines ensure that the status flags are updated following the execution of emulation routines which modify their values. The packing routines also update some of the intermediate variables which are used by the emulation kernel 70 and are sometimes called to update changes to these intermediate variables into status flags. The intermediate variables include conditional code register (CCR) values: Zero, eXtend, Negative, oVerflow and Carry flags CCR_Z, CCR_X, CCR_N, CCR_V and CCR_C respectively. The CCR variables are identified as a group of variables with the short form CCR_? where ? identifies the bit. The interrupt mask bit SR_I, the mode bit SR_S (supervisor/user) and the trace bit SR_T also exist. These variables are identified with the short form SR_? Where ? identified the bit. The bits are implemented as individual variables to reduce the processing required to support their functionality. To set, clear or test a variable requires less processing then having to isolate a given bit within a variable and then performing the same function on the resultant.

Referring now to Figure 26, the PackSr routine is better illustrated. As can be seen, when the PackSr routine is called, the conditional code variables CCR_? are updated and are given the status of their respective flags in Intel space (IntelSR) (block 270). The status register is then updated given the status of the SR_? bits and the CCR_? variables (block 272).

Figure 27 best illustrates the UnpackSR routine and as can be seen, when this routine is called, the stack pointer is firstly saved (block 280). The status register SR is then used to update the SR_? and CCR_? variables (block 282). Following this, the interruptMask is updated (block 284) and the Intel flags are used to update the IntelSR variable (block 286). The CCR bits in the IntelSR variable are then cleared (block 288) and the IntelSr variable is updated given the status of the CCR_? variables (block 290). The stack pointer is then retrieved (block 292) before the UnpackSr routine is exited.

Emulator Kernel

The M68k emulator kernel 56 utilizes a handful of internal variables which are important to its operation. These variables include the interrupt request, the interrupt mask, the current exception and the program counter. The interrupt request variable is utilized to determine which exception is set. The variable is of type "long" and each bit represents an exception, either set or clear. The exceptions fall within the following groupings: hardware interrupts (7-1 bit each), software exceptions (single bit) and system exceptions (4-bits). The hardware interrupts are supported by bits 0-6 of the least significant byte (LSB). Bit 0 represents interrupt 1 and bit 6 represents interrupt 7, the other interrupts are arranged hierarcally between these two extremes. The software exceptions are supported by bit 7 of the LSB. The bit only indicates whether or not a software exception has been requested. Information on the specific exception is contained in the current exception variable. The system exceptions are supported with 4 bits in the high byte of the least significant word and are set out in the table below.

Bit	Exception	Description
9	Termination	Utilized to support the Step functionality - performing a single instruction then returning to the system
13	DBErr	Double Bus Error
14	BGND	Background - used to exit the Emulation Kernel and return to the system temporarily then returning to the Kernel. This can be used to support M68k code with code running in native Intel x86.
15	Stop	Used to support the Stop functionality.

The interrupt mask is used to set the level of the current hardware interrupt being processed. This allows the interrupt hierarchy to be enforced. As in the M68k microprocessor, level 7 interrupt is not maskable. The variable saves the level number using the same bit positions as those used by the interrupt request variable. The variable is also used to support the trace functionality by enabling a level 7 mask whenever the trace is to be started.

The software exception is flagged by setting bit 7 of the interrupt request variable. The software exception that actually has been triggered is specified by the current exception variable. The value stored corresponds to the exception assignments used by Motorola to determine the exception vector. For example, vector 9 is used to indicate that a trace exception has been triggered or requested. Vector number 8 corresponds to a privilege violation etc.

The program counter variable is responsible for keeping track of the program flow by storing the position in memory at which the program is executing. Once an M68k instruction has been read from memory 72 in the main program loop, the program counter is augmented by 2 bytes (minimum size of operation). However, an operation may require that the program counter be modified further to the presence of data following the instruction.

Emulator API

Control Component 80

The control component as mentioned previously includes initialization, destroy, run and step functionality. Each is described in one of the following sections. The initialization routine is responsible for presetting the environment as required by the M68k emulator kernel 56 prior to its invocation. The destroy routine on the other hand is responsible for freeing and cleaning-up any resources used by the M68k emulator kernel 56 once the requirement for the same has expired.

The M68k emulator kernel 56 can be run in two different modes, namely a step mode or a run mode. The step mode is entered by invoking an Emu68kStep routine (see Figure 30). The Emu68kStep routine when executed also transfers control to the emulator kernel 70 and allows execution of a single M68k instruction. The M68k emulator kernel 56 is exited once the instruction has been processed. Although slower for regular processing, the Emu68kStep routine offers opportunities for debugging when combined with the other interfaces (memory, registers and exceptions). The Emu68kStep routine also allows for analyzer

functionality to be developed around the M68k emulator kernel 56. External looping with code or data breakpoints can be set up with minimum work.

To run a program, an Emu68kRun routine is invoked (see Figure 29). When the Emu68kRun routine is invoked, control is transferred to the emulator kernel 70 which in turn emulates M68k instructions utilizing the emulator loop 70. After each instruction is emulated, the program counter is adjusted and the following instruction is emulated. The M68k emulation kernel 56 is exited only to support a system exception as will be described. The continuous looping bypasses the overhead incurred to setup the emulating environment. Programs run using the Emu68kRun routine execute approximately 18 times faster than those that utilize the Emu68kStep routine in an external loop.

Initialization of the M68k emulator kernel 56 environment is required prior to operation. To initialize the M68k emulator kernel, an initialization Emu68kInitiazlize routine is called as shown in Figure 28. The Emu68kInitialize routine allocates the memory space utilized by the M68k emulator kernel by calling an Emu68kReserveMemory subroutine (block 300), presets some internal variables such as the status register, program counter, stack pointer, vector base register, and an internal mask for request values, initializes a vector table for support of exceptions in Intel space and allows for registration of support functions.

The M68k emulator kernel 56 provides for the ability to register functions to support particular operations. Specifically, functions can be registered to support a stop instruction, a background exception and a reset instruction. The stop instruction is supported as a default condition and is enabled unless a function is registered during initialization. In such a case, the new function is executed instead of the regular functionality (blocks 302 to 306). A routine is provided to take advantage of the stop instruction to exit gracefully from the M68k emulator kernel.

The background exception is provided to support an interface (a communication path) between the M68k code space and the Intel 80x86 native environment. The Emu68Initialize routine checks to see if the background exception

is supported. A stub is provided if the background exception is not implemented (blocks 308 to 312).

The reset instruction is used to trigger events in the Intel 80x86 native environment, hardware units, etc. The Emu68kInitialize routine also checks to see if the reset instruction is supported. A stub is provided if the reset instruction is not used (blocks 314 to 318).

Once the stop instruction, background exception and reset instruction have been supported, the M68k emulator kernel 56 executes an Emu68kInitExceptionsVectors routine (block 320). This routine offers the opportunity to bind to the M68k exception vectors and execute the bound functionality utilizing routines in Intel space. During initialization the vector table for these functions is cleared and initialized to a known state. Following this, an Emu68kSetupRequestMask routine and an Emu68kHwReset routine are executed (blocks 322 and 324)

The clean-up of the M68k emulator kernel environment is performed by an Emu68kDestroy routine. This routine releases the memory spaces allocated to the M68k emulator kernel 56 at initialization. The M68k emulator kernel does not instantiate separate threads and thus, none must be closed during clean-up.

Memory Access Component

During the start-up sequence of the M68k emulator kernel 56, the memory 72 is reserved and then the M68k instructions are loaded from file and stored into memory. The following sections describe the various functions available to interface to the M68k memory space.

The memory is reserved (and initialized to zero) during initialization of the M68k emulator kernel 56 by calling an Emu68kReserveMemory routine and is freed during clean-up/destruction by calling an Emu68kReleaseMemory routine. The memory space is contiguous from the point of view of the M68k code. The memory may not, in fact, be physically contiguous, but Windows NT performs the background

work to ensure that the virtual memory space as seen by the M68k emulator kernel 56 behaves as a contiguous section.

Amongst the differences in design of Motorola and Intel microprocessors one is of special concern, namely the big endian, little endian approach to memory usage. In the big endian scheme, data which must be stored using more than a single byte has its most significant byte stored first and the least significant byte stored last (with any intermediate bytes arranged in order between these two extremes). The little endian approach, of course, has the opposite arrangement. The least significant byte is stored first and the most significant byte is stored last. For both schemes, the bits of a byte are arranged identically. For example, a 32-bit integer with numerical value of 6, is represented by the bits 110 in byte 3 for a big endian scheme, but in byte 0 for a little endian scheme. Motorola microprocessors are based on the big endian scheme, while the Intel microprocessors use the little endian scheme.

The data is stored in memory 72 following the big endian scheme (Motorola) and is utilized by the M68k emulator kernel 56 in little-endian scheme. Within the M68k emulator kernel, all instructions which access memory areas are generated with byte swapping algorithms if required. Various routines are provided for external programs to deal with reading and writing information to memory and automatically to take care of the endian concern by performing byte swapping where necessary. These routines include an Emu68kReadByte routine, an Emu68kReadWord routine, an Emu68kReadLong routine, an Emu68kWriteByte routine, an Emu68kWriteWord routine and an Emu68kWriteLong routine.

To copy, store or retrieve large quantities of data, two other routines are provided, namely an Emu68kReadArray routine, and an Emu68kWriteArray routine. These routines do not provide compensation for the endian-concern. When dealing with structures, it is impossible to know the format of the data within.

To provide yet another tool, an Emu68kGetNtMemoryPtr routine is provided to interface with the M68k memory space directly. This routine returns a pointer in NT space to a required memory address within the M68k memory space.

Copying and writing directly from and to the M68k memory space is available via this routine.

In order to write M68k code to the memory 72, an Emu68kWriteSRecord routine is called. When called, the Emu68kWriteSRecord stores M68k code in the memory 72 using the S-Record format. Only S2 and S3 formats are supported by the M68k emulator kernel 56. The format specifies where within the memory the M68k code must be positioned. Multiple code segments may be written to memory by calling the routine repeatedly.

Figure 31 shows the Emu68kWriteSRecord routine and as can be seen, the routine employs subroutines to complete the work. The subroutines invoked by the Emu68k Write S-Record routine include an Emu68kProcessSRecord subroutine, an Emu68kGetByte subroutine, and a Emu68kWriteByte subroutine.

The Emu68kProcessSRecord subroutine is responsible for supporting the S-Record format and is best illustrated in Figure 32. This routine reads the bytes contained in the S-Records utilizing the Emu68kGetByte subroutine and interprets them. Once the data section of the S-Record is interpreted, the information is written to memory 72 using the Emu68kWriteByte subroutine.

Each time the Emu68kGetByte subroutine is used to determine a field (could be multiple calls generating a single value such as a long), a test is performed to check the validity of the data. Should the function indicate that an error condition occurred, the Emu68kProcessSRecord subroutine returns passing along an error code.

The Emu68kProcessSRecord subroutine also returns due to errors in the case of a checksum error (calculated does not match with value within record), in the case of bad data (error conditions during the Emu68kGetByte subroutine) or in the case where the address indicated in the S-Record exceeds the limit of the memory reserved during initialization.

Registers Component

The M68k emulator kernel 56 is designed around the concept of replacing an actual microprocessor. The interfaces provided use global variables. A

protection layer should be provided around the M68k emulator kernel to ensure that registers, memory and other components are not loosely modified. As mentioned previously, a variety of M68k registers are supported and available for use and include the program counter (PC), the status register (SR), the user and supervisor stack pointers (USP& SSP respectively), the data registers (Dn), the address registers (An), the vector base register (VBR), the source function code (SFC), and the destination function code (DFC). All of the registers are available as variables of size long (double word).

Requests Component

Requests (software exceptions and hardware interrupts) are handled within the emulator kernel 70. These consist of either hardware interrupts (set externally) or software exceptions (the SX2000 code relies on the current handlers contained within its load such as those prepared to support the Low Level Debugger functionality). All M68k exceptions are supported. However, for expediency in processing, the seven hardware interrupt exceptions are treated as separate entities.

Along with the status of the hardware interrupts, the InterruptRequest variable (long) indicates the status of software and system exceptions. The requests covered by the InterruptRequest variable include 7 levels of hardware interrupts (1 to 7), software exceptions (used to flag and process the software exceptions - traps, etc.), a termination exception (used to step through the code one instruction at a time), a stop exception, and a background exception (bgnd instruction is used as a system exception). Each exception and interrupt is represented by a bit in the InterruptRequest variable.

During initialization, a request mask is set-up to allow use of enumerated defines to refer to the various exceptions. This is used to provide a layer of abstraction from the code allowing more flexibility when updating or modifying code. The routine responsible for this is Emu68kSetupRequestMask.

In the M68k emulator kernel 56 there are effectively two types of requests, namely those that are available to anyone such as hardware interrupts and

the termination exception and those that are of a restricted nature and limited to the M68k emulator kernel 56 such as software exceptions in progress, a stop exception, double bus error, etc.

Various routines are provided to set, clear and test the requests. These routines include an Emu68kRequestSet routine, an Emu68kInternalRequestSet routine, an Emu68kRequestClear routine, an Emu68kInternalRequestClear routine, an Emu68kRequestTest routine which tests if the desired request has been set, an Emu68kInternalRequestTest routine, an Emu68kRequestTestEquality routine which tests if the desired request is the only one set, and an Emu68kInternalRequestTestEquality routine.

The "Emu68kInternal..." routines are those used internally by the M68k emulator 56 and are not limited on the request to be set, cleared or tested. The other routines, which are available for external use, use these internal routines to perform the functionality, but first call an Emu68kRestrictRequest routine to restrict access to the hardware interrupts and the termination exception. An error is returned if the function is attempted on a restricted request.

Figure 33 shows a sample flow chart and sample data flow diagram of an "Emu68kRequest..." routine.

Although the M68k emulator kernel 56 is designed around the concept of replacing an actual microprocessor, the M68k emulator provides some enhancements. In particular, the M68k emulator kernel 56 provides the opportunity to set (i.e. trigger) some of the base exceptions. The supported base exceptions are bus error (vector 2), address error (vector 3), illegal instruction (vector 4), divide by zero (vector 5), CHK instruction (vector 6), TRAPV instruction (vector 7), and privilege violation (vector 8).

The interface provided allows internal M68k emulator kernel variables to be modified thus triggering an exception. Once the exception has been generated (this occurs in NT space), the exception is processed as soon as the emulator kernel 70 has a chance to execute

Exceptions are set by passing the vector number (enumerated as above) to an Emu68kGenerateException routine. With a simple function call, the response to an address error, divide by zero, privilege violation, etc. can be tested without complicated steps in the emulated environment. An Emu68kRequestExitEmulator routine is also provided and sets a termination request and thus the termination of the main program loop. The routine can be called while servicing the background exception (or any other system exception that can be registered) to provide an organized exit form the emulator kernel 70.

The M68k emulator kernel environment provides possibilities not available in the native M68k microprocessor. The M68k emulator kernel can use the occurrence of exceptions in M68k space to trigger the execution of functions in NT space. This enables the exception to be debugged and/or logged in NT space. This has the advantage of being faster because code is executed in native Intel, and of becoming more integrated with the system using the emulation system.

Two routines are used to support binding exceptions, namely an Emu68kInitExceptionVectors routine, and an Emu68kBindExceptionVector routine.

Clearing and pre-setting the vector table occurs during initialization of the M68k emulator by calling an Emu68kInitExceptionVectors routine. This forces the vector table to a known state. Within the emulator kernel, this state is used to indicate that no routines have been bound and that normal M68k exception processing should be performed. A function can be bound to any (or many) given exception vector by calling an Emu68kBindExceptionVector routine. The routine will then be executed when the exception occurs and it will be processed before normal M68k exception processing.

Exceptions can be triggered externally to trace the execution flow or triggered internally to indicate that external processing is required. System exceptions exit from the M68k emulator kernel 56 (some temporarily) to the native system where they are processed and supported.

The M68k kernel 56 emulator supports a number of system exceptions, namely :

Termination (to perform a single instruction at a time and to exit emulator kernel);

Stop (support stop instruction);

DBErr (support a double bus error); and

Background (used to remove control from the M68k emulator kernel and return it to the calling function. This instruction has no relation to the BGND instruction of the Motorola 68040.

System Exception

Once a system exception is triggered, the emulator kernel 70 is exited and the Emu68kSystemExceptions routine (shown in Figure 34) is called to service the exception. A Debug exception is used to exit to the system. This system exception is only set by the Emu68kStep routine. It is used as a means of exiting the emulation kernel and returning to the system. The availability of this exception makes the step functionality available.

The stop exception is used to support "STOP" instructions. The stop instruction prevents further processing until a trace, interrupt or reset exception occurs. For interrupts, only those with a priority higher than that selected at the time the Stop instruction is called are considered.

The double-buss-error (DBErr) exception is used to indicate that a double bus error has occurred.

The background exception is used to exit from within the emulator kernel 70 and continue processing of specified functionality. The exception is triggered by a call from the M68k code to the bgnd instruction (introduced into emulated environment for this purpose). Each M68k instruction must be emulated by a multitude of 80x86 instructions and requires a proportionately high CPU time to execute when compared to a single native 80x86 instruction. It is an advantage whenever possible to process functionality outside the M68k emulator 56 and directly on the 80x86 platform. This includes both re-coding current functionality and writing implementations for new processes. Also a good reason to trigger external processing

is the possibility to interface with PC hardware or to take advantage of Windows NT abilities or tools to communicate with other processes and/or threads.

The Emu68kSystemExceptions routine calls a number of subroutines to support these exceptions. Specifically, the Emu68kSystemExceptions routine calls an Emu68kServiceDBErr subroutine to support double bus exceptions, calls an Emu68kServiceStop subroutine to support stop exceptions and calls an Emu68kServiceBackground subroutine to support background exceptions.

A reset instruction makes a direct call to an Emu68kSoftwareReset routine which in turn calls whichever function has been set-up to support the reset functionality (i.e. either the built-in stub function or a function registered during initialization).

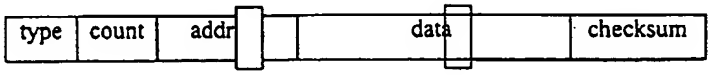
As will be appreciated by those of skill in the art, the emulator generator design is based on a table-driven philosophy allowing the overall design to be examined without getting lost in details. Detailed information however is readily available by inspecting the tables. The emulator generator 52 prepares the emulation routines to emulate each M68k instruction and prepares a jump-table that provides linking addresses to the emulation routines prior to running the M68k emulator kernel 56. When the M68k emulator kernel reads M68k instructions, it can quickly access and execute the corresponding emulation routines using the jump-tables to carry out the M68k instructions. This of course allows legacy code to be executed at high speeds while providing access to new tools, applications and functionality.

Although a preferred embodiment of the present invention has been described, those of skill in the art will appreciate that variations and modifications may be made without departing from the spirit and scope thereof as defined by the appended claims.

APPENDIX A

S-Record Format

An S-record file consists of a sequence of specially formatted ASCII character strings. An S-record will be less than or equal to 78 bytes in length. The order of S-records within a file is of no significance and no particular order may be assumed. The general format of an S-record follows:



Type	A char[2] field. These characters describe the type of record (S0, S1, S2, S3, S5, S7, S8 or S9)
Count	A char[2] field. These characters when paired and interpreted as an hexadecimal value, display the count of remaining character pairs in the record
Address	A char[4, 6, or 8] field. These characters grouped and interpreted as an hexadecimal value, display the address at which the data field is to be loaded into memory. The length of the field depends on the number of bytes necessary to hold the address. A 2-byte address uses 4 characters, a 3-byte address uses 6 characters, and a 4-byte address uses 8 characters.
Data	A char[0-64] field. These characters when paired and interpreted as hexadecimal values represent the memory loadable data or descriptive information.
Checksum	A char[2] field. These characters when paired and interpreted as an hexadecimal value display the least significant byte of the ones complement of the sum of the byte values represented by the pairs of characters making up the count, the address, and the data fields.

Each record is terminated with a line feed. If any additional or different record terminators(s) or delay characters are needed during the transmission to the target system it is the responsibility of the transmitting program to provide them.

S0 record	The type of the record is 'S0' (0x5330). The address field is unused and will be filled with zeros (0x0000). The header information within the data field is divided into the following subfields: mname - is char[20] and is the module name ver - is char[2] and is the version number rev - is char[2] and is the revision number description - is char[0-36] and is a text comment Each of the subfields is composed of ASCII bytes whose associated characters, when paired, represent one byte hexadecimal values in the case of the version and revision numbers, or represent the hexadecimal value of the ASCII characters comprising the module name and description.

APPENDIX A - CON'T

S1 record	The type of record is 'S1' (0x5331). The address field is interpreted as a 2-byte address. The data field is composed of memory lodable data.
S2 record	The type of record is 'S2' (0x5332). The address field is interpreted as a 3-byte address. The data field is composed of memory lodable data.
S3 record	The type of record is 'S3' (0x5333). The address field is interpreted as a 4-byte address. The data field is composed of memory lodable data.
S5 record	The type of record is 'S5' (0x5335). The address field is interpreted as a 2-byte value and contains the count of S1, S2 and S3 records previously transmitted. There is no data field.
S7 record	The type of record is 'S7' (0x5337). The address field contains the starting execution address and is interpreted as a 4-byte address. There is no data field.
S8 record	The type of record is 'S8' (0x5338). The address field contains the starting execution address and is interpreted as a 3-byte address. There is no data field.
S9 record	The type of record is 'S9' (0x5339). The address field contains the starting execution address and is interpreted as a 2-byte address. There is no data field.

Example:

```
S00600004844521B
S1130000285F245F2212226A000424290008237C2A
S11300100002000800082629001853812341001813
S113002041E900084E42234300182342000824A952
S107003000144ED492
S5030004F8
S9030000FC
```

The file consists of one S0 record, four S1 records, one S5 record and a single S9 record.

We Claim:

1. An emulation system to allow code written for a specific computing environment to be run on a different microprocessor platform comprising:

an emulation routine generator including a table storing code segments corresponding to instructions of said specific computing environment, said emulation routine generator creating said emulation routines written in code for said different microprocessor platform from selected code segments in said table, said emulation routines corresponding to instructions of said specific computing environment;

an emulator kernel to read instructions of said specific computing environment and access and execute emulation routines thereby to emulate and carry out said read instructions; and

an interface resembling said specific computing environment.

2. An emulation system as defined in claim 1 wherein said emulation routine generator creates an entry in a jump-table for each created emulation routine, each entry including linking addresses to said associated emulation routine.

3. An emulation system as defined in claim 2 wherein said emulation routine generator stores said emulation routines in an emulation code structure, said emulation code structure being divided in a plurality of assembly files.

4. An emulation system as defined in claim 3 wherein a nibble of each instruction is used to determine the assembly file of said emulation code structure into which the corresponding emulation routine is stored.

5. An emulation system as defined in claim 4 wherein entries in said jump-table are stored in a plurality of assembly files.

6. An emulation system as defined in claim 3 further including a data-table storing information describing the characteristics of all instructions of said specific computing environment, said information being accessed by said emulation routine generator and used to select code segments from said table when creating said emulation routines.

7. An emulation system as defined in claim 6 wherein said emulation routine generator includes an input component to access said data-table; an emulation routine generation component responsive to said input component to select code segments from said table; and an output component to piece said code segments together and write said code segments to said emulation code structure to form said emulation routines, said output component also creating said jump-tables.

8. An emulation system as defined in claim 7 wherein said emulator kernel is operable in a run mode, in said run mode said emulator kernel reading an instruction from said memory, executing the associated emulation routine and repeating the above steps for subsequent instructions in said memory.

9. An emulation system as defined in claim 8 wherein said interface is in the form of an emulator API.

10. An emulation system as defined in claim 9 wherein said interface includes a control component operable to condition said emulator kernel to said run and step modes.

11. An emulation system as defined in claim 10 wherein said interface further includes a requests component to generate said exceptions, a register component and a memory access component.

12. An emulator generator to generate emulation routines to be executed by an emulator running on one microprocessor platform to emulate instructions of a different microprocessor platform comprising:

a table storing code segments corresponding to instructions of said different microprocessor platform;

an emulation routine generation component creating emulation routines from selected code segments in said table, said emulation routine generation component further creating an entry in a jump-table for each created emulation routine, each said entry including linking addresses to said associated emulation routine.

13. An emulator generator as defined in claim 12 wherein said label is resolved into linking addresses for said emulation routine.

14. An emulator generator as defined in claim 13 wherein said emulation routine generator stores said emulation routines in an emulation code structure, said emulation code structure being divided in a plurality of assembly files.

15. An emulator generator as defined in claim 14 wherein a nibble of each instruction is used to determine the assembly file of said emulation code structure into which the corresponding emulation routine is stored.

16. An emulator generator as defined in claim 15 wherein entries in said jump-table are stored in a plurality of assembly files.

17. An emulator generator as defined in claim 16 further including a data-table storing information describing the characteristics of all instructions of said specific computing environment, said information being accessed by said emulation routine generator and used to select code segments from said table when creating said emulation routines.

18. An emulator generator as defined in claim 17 wherein said emulation routine generator includes an input component to access said data-table; an emulation routine generation component responsive to said input component to select code segments from said table; and an output component to piece said code segments together and write said code segments to said emulation code structure to form said emulation routines, said output component also creating said jump-tables.

The Patent Office

Application No:	GB 9819966.4	**Examiner:**	Mike Davis
Claims searched:	1-18	**Date of search:**	17 February 1999

Patents Act 1977
Search Report under Section 17

Databases searched:

UK Patent Office collections, including GB, EP, WO & US patent specifications, in:

UK Cl (Ed.Q): G4A (AFP)

Int Cl (Ed.6): G06F

Other:

Documents considered to be relevant:

Category	Identity of document and relevant passage		Relevant to claims
X	GB 2203572 A	(INSIGNIA SOLUTIONS)	1-18
X	GB 1593780	(NORAND)	1-18
X	GB 1497601	(HONEYWELL)	1-18
X	GB 1485874	(HONEYWELL)	1-18
X	US 4691278	(IWATA)	1-18

X	Document indicating lack of novelty or inventive step	A	Document indicating technological background and/or state of the art.
Y	Document indicating lack of inventive step if combined with one or more other documents of same category.	P	Document published on or after the declared priority date but before the filing date of this invention.
&	Member of the same patent family	E	Patent document published on or after, but with priority date earlier than, the filing date of this application.